CHAPTER 2

Visualizing Abstraction

> We have seen that computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty. A programmer who subconsciously views himself as an artist will enjoy what he does and will do it better.
>
> Donald E. Knuth
> *Turing Award Lecture (1974)*

> We may say most aptly that the Analytical Engine weaves algebraical patterns just as the Jacquard-loom weaves flowers and leaves.
>
> Ada Lovelace
> *Notes (1843)*

VISUALIZING large quantities of information can often provide insights that raw data cannot. Compare the following partial list of earthquake epicenters in (longitude, latitude) format with the visualization of these same data in Figure 2.1.

$$(-78.6, 19.3), (144.8, 19.1), (145.9, 43.5), (26.6, 45.7), (39.3, 38.4), (90.8, 26.3), \ldots$$

Simply plotting the points on an appropriate background provides immediate insight into recent seismic activity. A picture really is worth a thousand words, especially when we are faced with a slew of data.

This image was created with ***turtle graphics***. To draw in turtle graphics, we create an abstraction called a "turtle" in a window and move it with directional commands. As a turtle moves, its "tail" leaves behind a trail, as shown in Figure 2.2. If we lift a turtle's tail up, it can move without leaving a trace. In this chapter, in the course of learning about turtle graphics, we will also explore how abstractions can be created, used, and combined to solve problems.

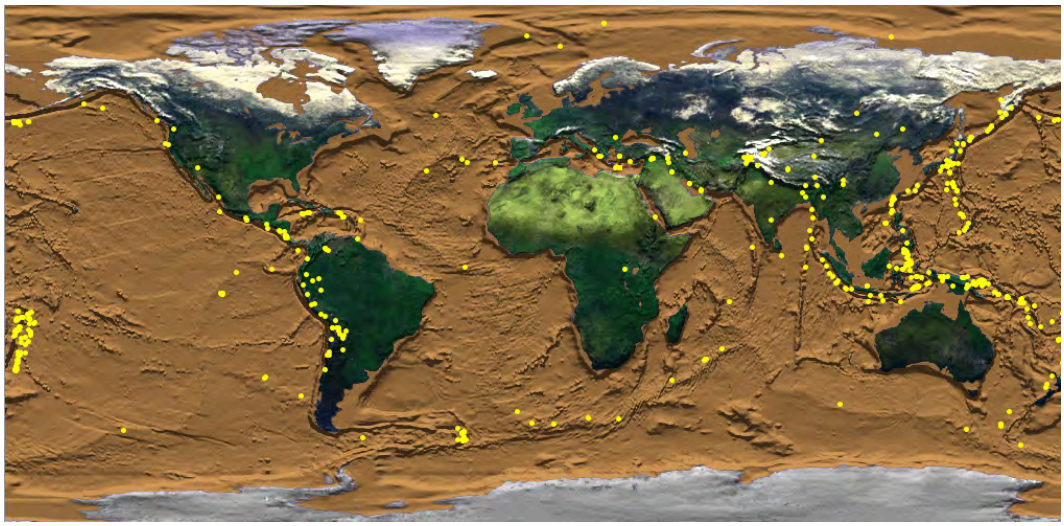


Figure 2.1 One month of magnitude 4.5+ earthquakes plotted.

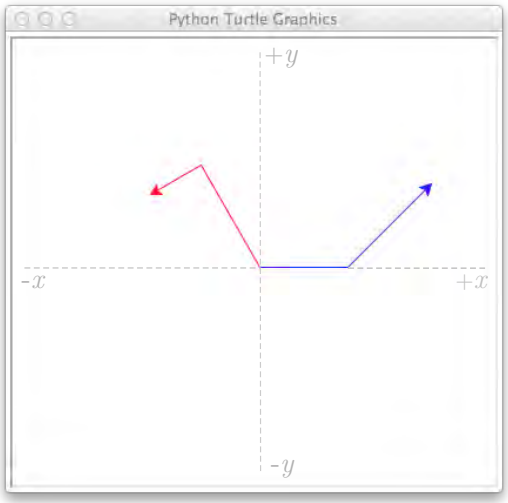


Figure 2.2 A turtle graphics window containing two turtles. The blue turtle moved forward, turned left 45°, and then moved forward again. The red turtle turned left 120°, moved forward, turned left again 90°, and then moved forward again.

2.1 DATA ABSTRACTION

The description of a turtle in turtle graphics is an example of an ***abstract data type*** (ADT). Just as a functional abstraction describes how to use a function or process without specifying how it works, an abstract data type describes how to use a category of *things* without necessarily specifying how they work. An ADT is composed of two parts:

(a) the types of information, called *attributes*, that we need to maintain about the things, and

(b) the *operations* that we are allowed to use to access or modify that information.

In Python, an abstract data type is implemented with a ***class***. In a class, attributes are maintained in a set of ***instance variables*** and the operations that can access these attributes are special functions called ***methods***.

The Python class that implements the Turtle ADT is named `Turtle`. The `Turtle` class contains several instance variables, some of which are listed below.

Instance Variable	Description
position	the turtle's current (x,y) position
heading	the turtle's current heading (in degrees)
color	the turtle's current drawing color
width	the turtle's current pen width
tail position	whether the turtle's tail is up or down

But we will never actually see or manipulate any of these instance variables directly. Instead, we will indirectly access and/or modify their values by calling the `Turtle` methods below.

Method	Argument	Description
forward	distance	move the turtle forward in its current direction
backward	distance	move the turtle opposite to its current direction
right, left	angle	turn the turtle clockwise or counterclockwise
setheading	angle	set the turtle's heading
goto	(x, y)	move the turtle to the given position
up, down	—	put the turtle's tail up or down
pensize	width	set the turtle's pen width
pencolor	color	set the turtle's pen color
position	—	return the turtle's (x,y) position
xcor, ycor	—	return the turtle's x or y coordinate
heading	—	return the turtle's heading

If the method requires an argument as input, that is listed in the second column of

the table. The first group of methods move the turtle, the second group change its other attributes, and the third group return information about its attributes. More `Turtle` methods are listed in Appendix A.2.

The `Turtle` class is defined inside a ***module*** named `turtle` (notice the different capitalization). A module is an existing Python program that contains predefined values and functions that you can use. To access the contents of a module, we use the `import` keyword.

```
>>> import turtle
```

After a module has been imported, we can access classes and functions in the module by preceding the name of thing we want with the name of the module, separated by a period (`.`). To confirm the existence of the `Turtle` class, try this:

```
>>> turtle.Turtle
<class 'turtle.Turtle'>
```

Just as a blueprint describes the structure of a house, but is not actually a house, the `Turtle` class describes the structure (i.e., attributes and methods) of a drawing turtle, but is not actually a drawing turtle. Actual turtles in turtle graphics, like those pictured in Figure 2.2, are called turtle ***objects***. An object is also called an *instance* of a class, hence the term *instance variable*. When we create a new turtle object belonging to the `Turtle` class, the turtle object is endowed with its own *independent* values of orientation, position, color, and so on, as described in the class definition. For this reason, there can be more than one turtle object, as illustrated in Figure 2.2.

The distinction between a class and an object can also be loosely described by analogy to animal taxonomy. A species, like a class, describes a category of animals sharing the same general (morphological and/or genetic) characteristics. An actual living organism is an instance of a species, like an object is an instance of a class. For example, the species of Galápagos giant tortoise (*Chelonoidis nigra*) is analogous to a class, while Lonesome George, the famous Galápagos giant tortoise who died in 2012, is analogous to an object of that class. Super Diego, another famous Galápagos giant tortoise, is a member of the same species but, like another object of the same class, is a distinct individual with his own unique attributes.

Reflection 2.1 *Can you think of another analogy for a class and its associated objects?*

Virtually any consumer product can also be thought of an object belonging to a class of products. For example, a pair of headphones is an object belonging to the class of all headphones with that particular make and model. The ADT or class specification is analogous to the user manual since the user manual tells you how to use the product without necessarily giving any information about how it works or how it is made. A course assignment is also analogous to an ADT because it describes the requirements for the assignment. When a student completes the assignment, she is creating an object that (hopefully) adheres to those requirements.

Turtle graphics

To create a turtle object in Python, we call a function with the class's name, preceded by the name of the module in which the class resides.

```
>>> george = turtle.Turtle()
```

The empty parentheses indicate that we are calling a function with no arguments. The `Turtle()` function returns a reference to a new `Turtle` object, which is then assigned to the name `george`. You should also notice that a window appears on your screen with a little arrow-shaped "turtle" in the center, facing east. The center of the window has coordinates (0,0) and is called the *origin*. In Figure 2.2, the axes are superimposed on the window in light gray to orient you to the coordinate system. We can confirm that `george` is a `Turtle` object by printing the object's value.

```
>>> george
<turtle.Turtle object at 0x100522f10>
```

The odd-looking "`0x100522f10`" is the address in memory where this `Turtle` object resides. The address is displayed in *hexadecimal*, or base 16, notation. The `0x` at the front is a prefix that indicates hexadecimal; the actual hexadecimal memory address is `100522f10`. If you're curious, see Tangent 3.3 in the next chapter for more about how hexadecimal works.

To call a method belonging to an object, we precede the name of the method with the name of the object, separated by a period. For example, to ask `george` to move forward 200 units, we write

```
>>> george.forward(200)
```

Since the origin has coordinates (0,0) and `george` is initially pointing east (toward positive x values), `george` has moved to position (200,0); the `forward` method silently changed `george`'s hidden position attribute to reflect this, which you can confirm by calling `george`'s `position` method.

```
>>> george.position()
(200.00,0.00)
```

Notice that we did not change the object's position attribute directly. Indeed, we do not even know the name of that attribute because the class definition remains hidden from us. This is by design. By interacting with objects only through their methods, and not tinkering directly with their attributes, we maintain a clear separation between the ADT specification and the underlying implementation. This allows for the possibility that the underlying implementation may change, to make it more efficient, for example, without affecting programs that use it. The formal term for this is ***encapsulation***, something we will discuss in more detail in Chapter 12.

Exercises

2.1.1. Explain the difference between an abstract data type and a Python class.

2.1.2* Design an ADT for a pair of wireless headphones using the same format we used to describe the `Turtle` ADT. Include attributes that describe the state of

the headphones at any given time and the operations that you can perform to change or get information about those attributes. You do not need to explain how to perform an operation, just what it does.

2.1.3. Choose an object from your everyday life and design an ADT for it using the format we used to describe the `Turtle` ADT.

2.1.4. Give another analogy for the difference between a class and an object. Explain.

2.1.5. Why do we use methods to change the state of a `Turtle` object instead of directly changing the values of its attributes?

2.1.6* In the Python shell, create a new turtle named `ada` and then turn `ada` 90 degrees clockwise like this:

```
>>> ada.right(90)
```

Use the `heading` method to show how the heading attribute of `ada` changed.

2.1.7. Create a new turtle named `gracie` and then move `gracie` like this:

```
>>> gracie.left(75)
>>> gracie.forward(200)
>>> gracie.right(150)
>>> gracie.forward(200)
>>> gracie.backward(80)
>>> gracie.right(105)
>>> gracie.forward(70)
```

How did these statements change the position and heading attributes of the turtle? Use the `position` and `heading` methods to find out.

2.1.8* Create two `Turtle` objects like this:

```
>>> thing1 = turtle.Turtle()
>>> thing2 = turtle.Turtle()
```

(a) What are the positions and headings of `thing1` and `thing2`? Use the `position` and `heading` methods.

(b) Using the `right` and `forward` methods, cause `thing2` to turn right 30 degrees and then move forward 50 units.

(c) What are the positions and headings of `thing1` and `thing2` now? Explain the values for each turtle and why they are different.

2.1.9. The following statements draw the red turtle in Figure 2.2.

```
>>> redTurtle = turtle.Turtle()
>>> redTurtle.pencolor('red')
>>> redTurtle.left(120)
>>> redTurtle.forward(100)
>>> redTurtle.left(90)
>>> redTurtle.forward(50)
```

Using this as an example and referring to the methods on page 51, create and draw the blue turtle in Figure 2.2 in the same window.

2.1.10. What is the difference between the statements `alice = turtle.Turtle` and `bob = turtle.Turtle()`? Which is the correct way to create a new `Turtle` object?

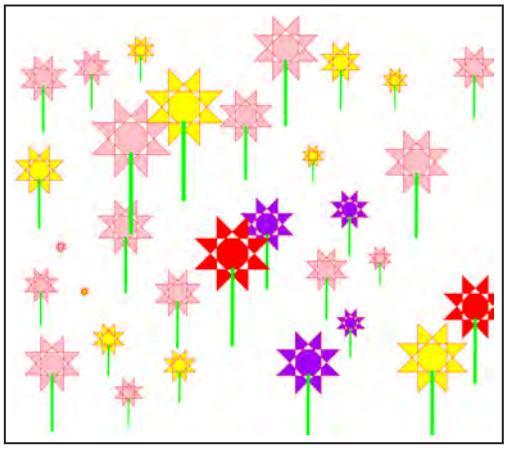


Figure 2.3 A garden of geometric flowers.

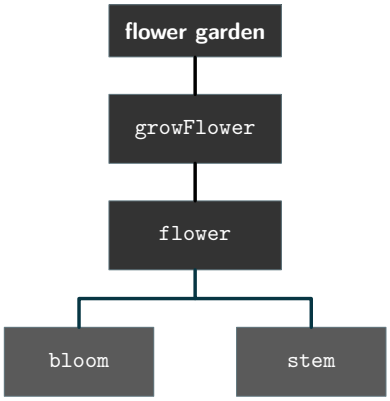


Figure 2.4 Functional decomposition of the flower garden problem.

2.2 DRAWING FLOWERS AND PLOTTING EARTHQUAKES

Before we look at how to plot the earthquakes in Figure 2.1, let's have some fun drawing flowers. Our ultimate goal, which we will complete in the next section, will be to plant a virtual garden of geometric "flowers" like those in Figure 2.3. To do this, we will implement the functional decomposition tree shown in Figure 2.4. An algorithm for the flower garden problem at the root of the tree will repeatedly call upon the `growFlower` algorithm to plant flowers at particular locations. The `growFlower` algorithm will choose the flower's size and color, and then call upon the `flower` algorithm to actually draw the flower. The `flower` algorithm is decomposed into two subproblems: one to draw the flower `bloom` and another to draw the `stem`. In our bottom-up implementation of this design, we will work our way up from the leaves of the tree toward the root, starting with the `bloom` subproblem, which will draw geometric flower bloom in Figure 2.5.

To start the bloom, we can use the line we drew in the last section as the lower horizontal line segment in the figure. Before we draw the next segment, we need to ask `george` to turn left 135 degrees. This line is highlighted below, following the three steps from the last section, in case you need to type them again.

```
>>> import turtle
>>> george = turtle.Turtle()
>>> george.forward(200)
>>> george.left(135)
```

With this method call, we have changed `george`'s hidden heading attribute, which we can confirm by calling the `heading` method.

```
>>> george.heading()
135.0
```

To finish the drawing, we just have to repeat the previous `forward` and `left` calls seven more times! (Hint: see IDLE help for how retrieve previous statements.)

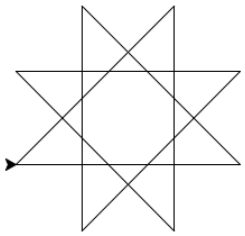


Figure 2.5 A simple geometric flower bloom drawn with turtle graphics.

```
>>> george.forward(200)
>>> george.left(135)
>>> george.forward(200)
>>> george.left(135)
>>> george.forward(200)
>>> george.left(135)
>>> george.forward(200)
>>> george.left(135)
>>> george.forward(200)
>>> george.left(135)
>>> george.forward(200)
>>> george.left(135)
>>> george.forward(200)
>>> george.left(135)
```

That was tedious. But before we look at how to avoid similar tedium in the future, we are going to transition out of the Python shell. This will allow us to save our programs so that we can easily modify them or fix mistakes, and then re-execute them without retyping everything. In IDLE, we can create a new, empty program file by choosing New Window from the File menu.[1] In the new window, retype (or copy and paste) the work we have done so far, plus the four additional highlighted lines, shown below. (If you copy and paste, be sure to remove the >>> characters.)

```
import turtle

george = turtle.Turtle()
george.hideturtle()
george.speed(6)

george.forward(200)
george.left(135)
george.forward(200)
george.left(135)
george.forward(200)
george.left(135)
george.forward(200)
george.left(135)
```

[1] If you are using a different text editor, the steps are probably very similar.

```
george.forward(200)
george.left(135)
george.forward(200)
george.left(135)
george.forward(200)
george.left(135)
george.forward(200)
george.left(135)

screen = george.getscreen()
screen.exitonclick()
```

The two highlighted statements after the first assignment statement hide `george` and speed up the drawing a bit. (The argument to the `speed` method is a number from 0 to 10, with 1 being slow, 10 being fast, and 0 being fastest.) The second to last statement assigns to the variable `screen` an object of the class `Screen`, which represents the drawing area in which `george` lives. The last statement calls the `Screen` method `exitonclick` which will close the window when we click on it. These last two lines are only necessary if your programming environment closes the turtle graphics window when the program is done. If it does not, you can omit these lines and close the window yourself when you are finished. In the future, we will generally leave these lines out, but feel free to include them in your programs as desired.

When you are done, save your file by selecting Save As... from the File menu. The file name of a Python program must always end with the extension `.py`, for example, `george.py`. To execute your new program in IDLE, select Run Module from the Run menu. If IDLE prompts you to save your file again, just click OK. After the program draws the flower, click on the turtle graphics window to dismiss it.

Iteration

Recall from Chapter 1 that we can use ***loops*** in algorithms to repeat statements multiple times, a process called ***iteration***.

Reflection 2.2 *In pseudocode, how could we use a loop to simplify this long sequence of statements?*

Since we have eight identical pairs of calls to the `forward` and `left` methods, we can replace these sixteen drawing statements with a loop that repeats one pair eight times. In pseudocode, this would look something like this:

```
repeat the following eight times:
    george.forward(200)
    george.left(135)
```

In Python, we can use a `for` loop, inserted into our program below.

```
1 import turtle

2 george = turtle.Turtle()
3 george.hideturtle()
4 george.speed(6)

5 for count in range(8):
6     george.forward(200)
7     george.left(135)

8 screen = george.getscreen()
9 screen.exitonclick()
```

This `for` loop, in lines 5–7, repeats the indented statements, called the ***body*** of the loop, eight times. After the loop is done, the next non-indented statement on line 8 is executed.

Reflection 2.3 *What happens if you forget the colon at the end of line 5? (Try it.)*

It is easy to forget the colon. If you do, you will be notified by a syntax error, like the following, that points to the end of the line containing the `for` keyword.

```
for count in range(8)
                     ^
SyntaxError: invalid syntax
```

In the `for` loop syntax, `for` and `in` are Python keywords, and `count` is called the ***index variable***. The name of the index variable can be anything we want, but it should be descriptive of its role in the program. In this case, we chose the name `count` because it is counting the number of line segments that are being drawn. The part after the `in` keyword is a sequence of some kind:

```
for count in range(8):
    index variable  sequence
```

At the beginning of each iteration of the loop, the next value in the sequence is assigned to the index variable, and then the statements in the body of the loop are executed. In this case, `range(8)` represents the sequence of eight integers from 0 to 7. So this `for` loop is saying

> For each number in the range from 0 to 7, assign the number to `count`, and then execute the body of the loop.

The following trace table shows the execution of the program in more detail. The value of `george` is represented in the trace table by an image of what has been drawn so far in the program.

Trace

Step	Line	george	count	Notes
1–4	1–4		—	initialize `george`
5	5	″	0	`count = 0`
6	6		″	`george.forward(200)`
7	7		″	`george.left(135)`
8	5	″	1	`count = 1`
9	6		″	`george.forward(200)`
10	7		″	`george.left(135)`
11	5	″	2	`count = 2`
12	6		″	`george.forward(200)`
13	7		″	`george.left(135)`
⋮				
26	5	″	7	`count = 7`
27	6		″	`george.forward(200)`
28	7		″	`george.left(135)`
29	8	″	″	`screen = george.getscreen()`
30	9	—	″	`screen.exitonclick()`

After the program initializes the turtle graphics window in lines 1–4, the `for` loop is reached on line 5. In the first iteration, `count` is assigned the first value in the range of numbers from 0 to 7. Then the body of the loop in lines 6–7 is executed. Once the body of the loop is complete, we return to line 5 (step 8) to execute the second iteration of the loop. This time, `count` is assigned 1 and the body of the loop is executed again. This continues for six more iterations since there are six more values in the range from 0 to 7 for `count` to be assigned. After all eight iterations of the loop are complete, the last two lines in the program are executed.

Tangent 2.1: Defining colors

The most common way to specify an arbitrary color is to specify its red, green, and blue (RGB) components individually. Each of these components is often described by an integer between 0 and 255. (These are the values that can be represented by 8 bits. Together then, a color is specified by 24 bits. If you have heard a reference to "24-bit color," now you know its origin.) Alternatively, each component can be described by a real number between 0 and 1.0. In Python turtle graphics, call `screen.colormode(255)` or `screen.colormode(1.0)`, where `screen` is a turtle's `Screen` object, to choose the desired representation.

A higher value for a particular component represents a brighter color. So, at the extremes, (0,0,0) represents black, and (255,255,255) and $(1.0, 1.0, 1.0)$ both represent white. Other common colors include $(255, 255, 0)$ for yellow, $(127, 0, 127)$ for purple, and $(153, 102, 51)$ for brown. So, assuming `george` is a `Turtle` object and `screen` has been assigned `george`'s `Screen` object,

```
screen.colormode(255)
george.pencolor((127, 0, 127))
```

would make `george` purple. So would

```
screen.colormode(1.0)
george.pencolor((0.5, 0, 0.5))
```

Reflection 2.4 *Try different values between* `1` *and* `10` *in place of* `8` *in* `range(8)`. *Can you see the connection between the value and the picture?*

Another way to see what is happening in this loop is to print the value of `count` in each iteration. To do this, add `print(count)` to the body of the `for` loop:

```
for count in range(8):
    george.forward(200)
    george.left(135)
    print(count)
```

Now, in each iteration, `george` is drawing a line segment and turning left, and then the current value of `count` is printed. As you run the program, you should notice that the numbers 0 through 7 are printed in the shell as the eight line segments are drawn in the turtle graphics window.

Reflection 2.5 *Try changing* `count` *to some other name. Did changing the name change the behavior of the program? If you changed the name only in the* `for` *loop and not in the* `print` *statement, you will get an error because* `count` *will no longer exist! You need to change it to the same thing in both places because the variable in the* `print` *statement refers to the index variable in the* `for` *loop.*

Adding some color

To finish up the bloom, let's add some color. To set the color that the turtle draws in, we use the `pencolor` method. Insert

```
george.pencolor('red')
```

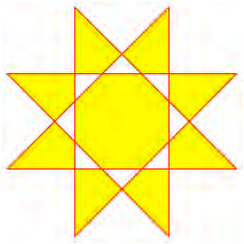


Figure 2.6 A simple geometric flower bloom, outlined in red and filled in yellow.

before the `for` loop, and run your program again. A color can be specified in one of two ways. First, common colors can be specified with strings such as `'red'`, `'blue'`, and `'yellow'`. Remember that a string must be enclosed in quotes to distinguish it from a variable or function name. A color can also be defined by explicitly specifying its red, green, and blue (RGB) components, as explained in Tangent 2.1.

Finally, we will specify a color with which to fill the "flower" shape. The fill color is set by the `fillcolor` method. The statements that draw the area to be filled must be contained between calls to the `begin_fill` and `end_fill` methods. To color our flower yellow, precede the `for` loop with

```
george.fillcolor('yellow')
george.begin_fill()
```

and follow the `for` loop with

```
george.end_fill()
```

Be sure to *not* indent the call to `george.end_fill()` in the body of the `for` loop since we want that statement to execute just once *after* the loop is finished. Your flower should now look like Figure 2.6, and the complete flower bloom program should look like the following:

```
import turtle

george = turtle.Turtle()
george.hideturtle()
george.speed(6)

george.pencolor('red')
george.fillcolor('yellow')
george.begin_fill()
for count in range(8):
    george.forward(200)
    george.left(135)
george.end_fill()

screen = george.getscreen()
screen.exitonclick()
```

Reflection 2.6 *Can you figure out why the shape was filled this way?*

In the next section, we will put some finishing touches on our flower bloom and flesh out the decomposition tree in Figure 2.4. But first, let's return to the earthquake visualization from Figure 2.1 at the beginning of the chapter.

Data visualization

To create the earthquake visualization, we want to draw a dot at each earthquake location in a list, so our pseudocode algorithm might look like this:

> repeat for each earthquake *location* in a list:
> draw a dot at the *location*

In Python, drawing a dot is actually a two step process. First, we have to move the turtle to the location and then draw a dot there. The moving part is accomplished by the `goto` method. For example,

```
george.goto(150, 30)
```

moves `george` to the coordinates $(150, 30)$ in the turtle graphics window. Once there, we can draw a dot with

```
george.dot()
```

To implement the loop in Python, we will use a `for` loop that iterates over a list of earthquake locations like `(-78.6, 19.3)` rather than over a range of numbers. The first value in each ordered pair is the earthquake's longitude and the second value is the latitude. A list of these locations looks like this:

```
quakes = [(-78.6, 19.3), (144.8, 19.1), (145.9, 43.5), (26.6, 45.7)]
```

The variable `quakes` is being assigned a ***list*** of ordered pairs.[2] A list in Python is always surrounded by square brackets (`[ ]`). This list only contains the locations of four earthquakes; the full program with a longer list is available on the book website. To iterate over this list, we use a `for` loop with the list `quakes` as the sequence:

```
for location in quakes :
```

`location`: index variable; `quakes`: sequence

In the body of the `for` loop, we will pass the index variable `location` to the `goto` method and then draw a dot with the `dot` method. The full program follows, with some extra pretty formatting. The core plotting statements are highlighted.

[2] Each ordered pair is actually called a ***tuple*** in Python. We will see tuples in more detail in Chapter 7.

```
1 import turtle

2 george = turtle.Turtle()
3 screen = george.getscreen()
4 screen.setup(1024, 512)
5 screen.bgpic('oceanbottom.gif')
6 screen.setworldcoordinates(-180, -90, 180, 90)

7 george.speed(0)
8 george.hideturtle()
9 george.up()
10 george.color('yellow')

11 quakes = [(-78.6, 19.3), (144.8, 19.1), (145.9, 43.5), (26.6, 45.7)]

12 for location in quakes:
13     george.goto(location)
14     george.dot()
```

As with the flower `for` loop, we can illustrate what is happening in more detail with a trace table. We represent the value of `george` with an image of what the screen looks like at each point in the program.

Trace

Step	Line	george	location	Notes
1–10	1–10		—	initialize george and the window
11	11	″	—	quakes = a list of ordered pairs
12	12	″	(-78.6, 19.3)	location = (-78.6, 19.3)
13	13		″	george.goto((-78.6, 19.3))
14	14	″	″	george.dot()
15	12	″	(144.8, 19.1)	location = (144.8, 19.1)
16	13		″	george.goto((144.8, 19.1))
17	14	″	″	george.dot()
18	12	″	(145.9, 43.5)	location = (145.9, 43.5)
19	13		″	george.goto((145.9, 43.5))
20	14	″	″	george.dot()
21	12	″	(26.6, 45.7)	location = (26.6, 45.7)
22	13		″	george.goto((26.6, 45.7))
23	14	″	″	george.dot()

The first ten lines set up the drawing window with a background picture of the earth. The `Screen` method `setworldcoordinates` is used to set the coordinate system inside the window to match geographical coordinates: longitude values run from

−180 to 180 and latitude values run from −90 to 90. The first two arguments set the bottom left corner of the window to be $(-180, -90)$ and the last two arguments set the top right corner to be $(180, 90)$. We encourage you to consult Appendix A.3 to learn what the other methods do.

In the first iteration of the `for` loop on line 12, the index variable `location` is assigned the first pair in the list `quakes`, which is `(-78.6, 19.3)`. Next, `george.goto(location)` is executed; since `location` was assigned `(-78.6, 19.3)`, this is equivalent to `george.goto((-78.6, 19.3))`. Then a dot is drawn at that location. Once the first iteration is complete, we return to line 12 for the second iteration, where `location` is assigned the second pair in the list, which is `(144.8, 19.1)`. Lines 13–14 are executed again, which draws a dot at this location. This process continues for two more iterations since there are two more pairs remaining in the list.

Reflection 2.7 *What is the significance of* `george.up()` *in the program? What happens if you omit it?*

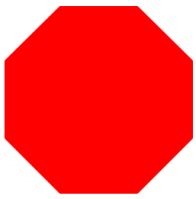
You can download `oceanbottom.gif` and this complete program with more earthquake locations from the book website.

Exercises

Write a short program to answer each of the following questions. Submit each as a separate python program file with a `.py` *extension (e.g.,* `picture.py`*).*

2.2.1. Write a program using turtle graphics that draws a national or state flag of your choice. You might want to consult `https://en.wikipedia.org/wiki/Gallery_of_sovereign_state_flags` for ideas and Appendices A.2 and A.3 for additional drawing methods.

2.2.2. Write a program that draws the following three shapes (resembling street signs) using turtle graphics.

(a)*  (b)  (c) 

2.2.3. Draw an interesting picture using turtle graphics. Consult Appendices A.2 and A.3 for a list of methods. You might want to draw your picture on graph paper first.

2.2.4. Modify the `for` loop in the flower bloom program so that it draws a flower with 18 line segments, using an angle of 100°.

2.2.5* Write a program that uses a `for` loop to draw a square with side length 200.

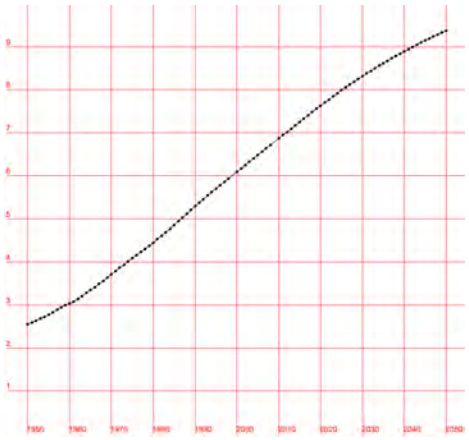
2.2.6. Write a program that uses a `for` loop to draw a rectangle with length 200 and width 100.

2.2.7. Write a program that instructs a turtle to repeat the following 180 times: draw forward 200, return to the origin, turns 2 degrees left.

2.2.8* Suppose you have the coordinates of discovered artifacts from a 9 meter × 9 meter plot during an archaeological dig. The coordinates extend from (0,0) in the bottom left corner of the plot to (9,9) in the upper right corner. Write a program that plots a list of these coordinates to detect any patterns in their locations. The list of coordinates is available on the book website.

2.2.9. Write a program that uses turtle graphics to draw a line graph of the world population from 1950 to 2050 (projected). The data is stored in a list of (year, population) pairs, available on the book website. The population is recorded in billions, e.g., 2.5 represents 2.5 billion. Use `setworldcoordinates` to set the bottom left corner of the window to be (1945,0) and the top right corner to be (2055,10). You can draw vertical lines to mark the years in your plot with the following loop:

```
for year in range(1950, 2051, 10):
    george.up()
    george.goto(year, 0)
    george.write(str(year))
    george.down()
    george.goto(year, 10)
```



This `range` is equivalent to the list $1950, 1960, \ldots, 2050$. As `year` is assigned to each of these values in the loop, a vertical line is drawn from bottom to top. The `write` method prints the year, converted to a string, at the bottom of each line. Using this as a model, also draw horizontal lines to mark each billion. Your final graph should look like that to the right.

2.2.10. Suppose an ant is moving in a straight line toward its nest three meters away. A hungry fly, exactly one meter above the ant, starts to fly directly toward it at exactly the same speed. Will the fly catch the ant? If not, how close will it come?

We can write a turtle graphics simulation of this scenario by modeling each insect as a turtle. The ant turtle starts at position (0,0) and the fly turtle starts at position (0,1), directly above the ant. At each step, the fly moves one step directly toward the ant and the ant moves one step forward. The length of each step is the distance to the nest divided by the total number of steps that we want the insects to take. The more steps they take, the closer the simulation becomes to a continuous real-life scenario.

Implement this simulation, based on the following pseudocode algorithm.

Algorithm THE ANT AND THE FLY

Input: none

```
1  nest distance ← 3
2  total steps ← 300
3  step length ← nest distance ÷ total steps
4  create the ant at position (0,0)
5  create the fly and move it to position (0,1)
6  repeat total steps times:
7      turn the fly toward the ant
8      move the fly forward step length
9      move the ant forward step length
```

Output: the final distance between the ant and the fly

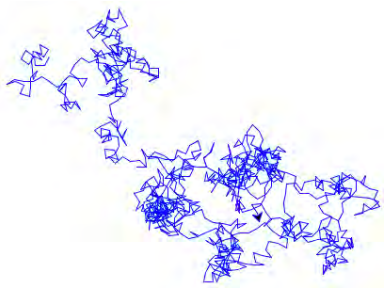
In your program, before you move any turtles, use `setworldcoordinates` to set the bottom left corner of the window to be (0,0) and the top right corner to be (*nest distance*, 1) so that you can see what is happening much more clearly. Also, there are two new `Turtle` methods that will make your job easier. The methods `towards` and `distance` return the angle and distance, respectively, between the turtle and another turtle. For example, `fly.towards(ant)` will return the angle between the fly turtle and the ant turtle (assuming you have named them `fly` and `ant`, of course). The `distance` method works similarly. At the end of your program, print the final distance between the two insects. Does the result surprise you?

2.2.11. A random walk simulates a particle, or person, randomly moving in a two-dimensional space. At each step, the particle turns in a random direction and walks a fixed distance (e.g., 10 units) in the current direction. If this step is repeated many times, it can simulate Brownian motion or animal foraging behavior.

Write a program that uses turtle graphics to draw a 1000-step random walk. To randomly choose an angle in each step, use

```
angle = random.randrange(360)
```

You will need to import the `random` module to use this function. (We will talk more about random numbers and random walks in Chapter 5.) One particular 1000-step random walk is shown to the right.



2.3 FUNCTIONAL ABSTRACTION

To draw the garden of flowers from Figure 2.3, each with a different color and size, we are going to need to repeat our flower bloom code many times. We could do this by copying the drawing statements and changing method arguments to

alter the sizes and colors. However, this strategy is a *very bad idea.* First, it is very time-consuming and error-prone; when you repeatedly copy and paste, it is very easy to make mistakes. Second, it makes your program unnecessarily long and hard to read. Third, it is difficult to correctly make changes. For example, what if you copied enough to draw twenty flowers, and then decided that you wanted to give all of them six petals instead of eight?

Instead, we want to create a self-contained functional abstraction that will draw a flower bloom when called upon to do so. In Python, we do this by creating a new function. Functions are like our pseudocode algorithms in that they can take inputs, produce outputs, and can be called upon by other algorithms to perform tasks. To create a function in Python, we use the `def` keyword, followed by the function name and, for now, empty parentheses (we will come back to those shortly). As with a `for` loop, the `def` line must end with a colon (`:`).

```
def bloom():
```

The body of the function is then indented relative to the `def` line. The body of our new function will consist of the flower bloom code. Insert this new function into your program from the last section after the `import` statement:

```
import turtle

def bloom():
    george.pencolor('red')
    george.fillcolor('yellow')
    george.begin_fill()
    for count in range(8):
        george.forward(200)
        george.left(135)
    george.end_fill()

george = turtle.Turtle()
george.hideturtle()
george.speed(6)

bloom()

screen = george.getscreen()
screen.exitonclick()
```

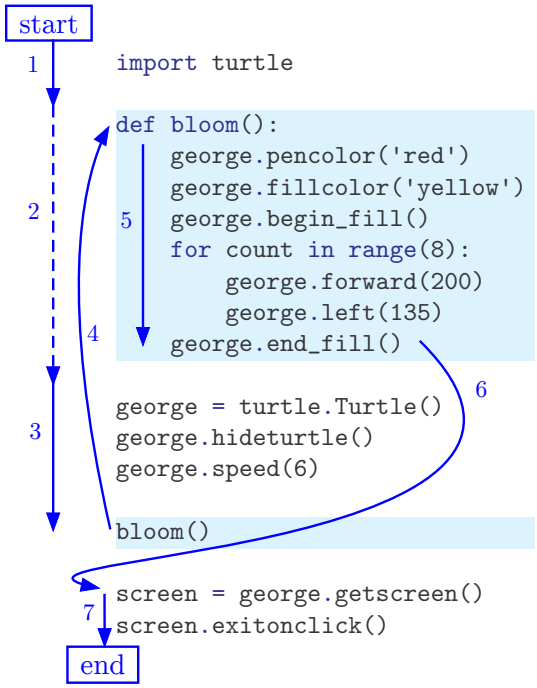
The `def` construct *only defines* the new function; it *does not execute* it. We need to *call* the function for it to execute. As we saw earlier, a function call consists of the function name, followed by a list of arguments. Since this function does not have any arguments (yet), and does not return a value, we can call it with

```
bloom()
```

inserted, at the outermost indentation level, where the flower bloom code used to be (as shown above).

Reflection 2.8 *Try running the program with and without the* `bloom()` *function call. What happens in each case?*

Before continuing, let's take a moment to look closely at what the program is doing. As illustrated below, execution begins at the top, labeled "start." After that, there are seven labeled steps, explained below.



1. Import the `turtle` module.
2. Define the `bloom` function. Note that the function is *not* executed yet; Python is just learning of its existence so that it can be called later.
3. The next three statements *are* executed. They define a new `Turtle` object named `george`, hide the turtle, and speed it up a bit.
4. Next, the `bloom()` function is called, which causes execution to jump up to the beginning of the function.
5. The statements in the function then draw the flower.
6. When the function is complete, execution continues with the statement after the function call.
7. And the program ends.

Function parameters

The `bloom` function is not as useful as it could be because it always draws the same yellow flower with segment length 200. We can generalize the function by accepting the fill color and the segment length as arguments, as depicted below.

We do this by adding ***parameters*** to the function definition. A parameter is the name of an input, like the inputs in our pseudocode algorithms. In the highlighted lines of the new version below, we have defined two parameters in parentheses after the function name to represent the fill color and the segment length, and replaced the old constants `'yellow'` and `200` with the names of these new parameters.

```
import turtle

def bloom(color, length):
    george.pencolor('red')
    george.fillcolor(color)
    george.begin_fill()
    for count in range(8):
        george.forward(length)
        george.left(135)
    george.end_fill()

george = turtle.Turtle()
george.hideturtle()
george.speed(6)

bloom('yellow', 200)

screen = george.getscreen()
screen.exitonclick()
```

To replicate the old behavior, we added two arguments to the function call:

```
bloom('yellow', 200)
```

When this function is called, the value of the first argument `'yellow'` is assigned to the first parameter `color` and the value of the second argument `200` is assigned to the second parameter `length`. Then the body of the function executes. Whenever `color` is referenced, it is replaced with `'yellow'`, and whenever `length` is referenced, it is replaced with `200`. (Parameters and arguments are also called *formal parameters* and *actual parameters*, respectively.)

Reflection 2.9 *After making these changes, run the program again. Then try running it a few more times with different arguments passed into the* `bloom` *function call. For example, try* `bloom('orange', 50)` *and* `bloom('purple', 350)`*. What happens if you switch the order of the arguments in one these function calls?*

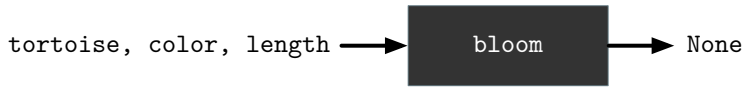
We are going to make one more change to this function before moving on, motivated by the following question.

Reflection 2.10 *Look at the variable name* `george` *that is used inside the* `bloom` *function. Where is it defined?*

When the `bloom` function executes, the Python interpreter encounters the variable name `george` in the first line, but `george` has not been defined in that function. Realizing this, Python looks for the name `george` outside the function. This behavior is called a ***scoping rule***. The ***scope*** of a variable name is the part of the program where the name is defined, and hence can be used.

The scope of a variable name that is defined inside a function, such as `count` in the `bloom` function, is limited to that function. Such a variable is called a ***local variable***. If we tried to refer to `count` outside of the the `bloom` function, we would get an error. We will look at local variables in more detail in Section 2.6.

A variable name that is defined at the outermost indentation level can be accessed from anywhere in the program, and is called a ***global variable***. In our program, `george` and `screen` are global variable names. It is generally a bad idea to have any global variables at all in a program, a topic that we will further discuss in the next sections. But even aside from that issue, we should be concerned that our function is tied to one specific turtle named `george` that is defined outside our function. It would be much better to make the turtle a parameter to the function, so that we can call it with any turtle we want, as illustrated below:



Replacing `george` with a parameter named `tortoise` gives us the following modified function:

```
def bloom(tortoise, color, length):
    tortoise.pencolor('red')
    tortoise.fillcolor(color)
    tortoise.begin_fill()
    for count in range(8):
        tortoise.forward(length)
        tortoise.left(135)
    tortoise.end_fill()
```

We also need to update the function call by passing `george` as the first argument, to be assigned to the first parameter, `tortoise`.

```
bloom(george, 'yellow', 200)
```

Now that the bloom is finished, we need to create a function that draws a stem. Our stem-drawing function will take two parameters: `tortoise`, which is the name of the turtle object, and `length`, the length of the stem.

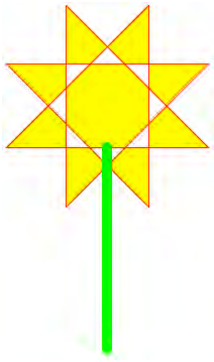


Figure 2.7 A simple geometric "flower" with a stem.

In the following function, notice all the places, highlighted in red, where the parameters are being used.

```
1 def stem(tortoise, length):
2     tortoise.pencolor('green')
3     tortoise.pensize(length / 20)
4     tortoise.up()
5     tortoise.forward(length / 2)
6     tortoise.down()
7     tortoise.right(90)
8     tortoise.forward(length)
```

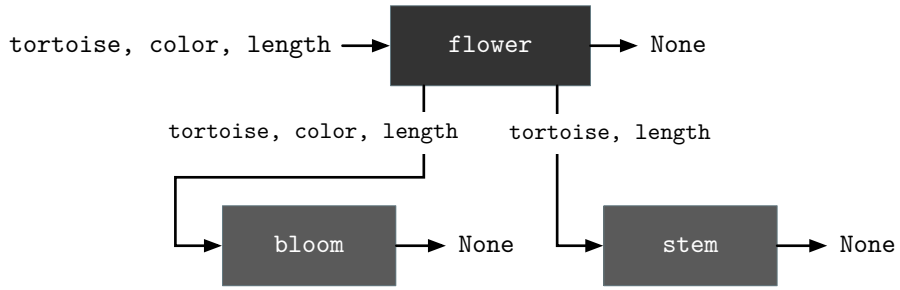
For convenience, we assume that the stem length is the same as the length of a segment in the associated flower. Since the `bloom` function nicely returns the turtle to the origin, pointing east, we will assume that `tortoise` is in this state when `stem` is called. We start the function by setting the pen color to green, and thickening the turtle's tail by calling the method `pensize`. Notice that the pen size on line 3 is based on the parameter `length`, so that it scales properly with different size flowers. Next, in lines 4–6, we move halfway across the flower to start drawing the stem. So that we do not draw over the existing flower, we put the turtle's tail up with the `up` method before we move, and return it to its resting position again with `down` when we are done. Finally, in lines 7–8, we turn to the south and move the turtle forward to draw a thick green stem.

To draw a stem for our yellow flower, insert this function in your program after where the `bloom` function is defined, and then call it with

```
stem(george, 200)
```

after the call to the `bloom` function. When you run your program, the flower should look like Figure 2.7.

We now have functions—functional abstractions—that implement the two subproblems of the `flower` problem from our decomposition in Figure 2.4. So we are ready to use these to create a function (another functional abstraction) that draws a flower, as depicted below.



Because the `bloom` and `stem` functions together require a turtle, a fill color and a length, and we want to be able to customize our flower in these three ways, these are the parameters to our `flower` function. We pass all three of these parameters through to the `bloom` function, and then we pass two of them to the `stem` function. In Python, our function looks like this:

```
def flower(tortoise, color, length):
    bloom(tortoise, color, length)
    stem(tortoise, length)
```

A complete program incorporating these functions is shown in Figure 2.8.

Exercises 2.3.6–2.3.8 below challenge you to implement the remaining layers of the decomposition tree to create a full garden of flowers, as illustrated in Figure 2.3.

Exercises

Write a short program to answer each of the following questions. Submit each as a separate python program file with a `.py` *extension (e.g.,* `picture.py`*).*

2.3.1. Modify the program in Figure 2.8 so that it calls the `flower` function three times to draw three flowers, each with a different color and size. You will want to move the turtle and reset its pen size and heading to their original values before drawing each flower so that they are drawn correctly and not on top of each other.

2.3.2* Modify the `bloom` function so that it draws 10 petals instead of 8. In each iteration of the loop, the turtle will need to turn 108 degrees instead of 135.

2.3.3. Modify the `bloom` function so that it can draw any number of petals. The revised function will need to take an additional parameter:

```
bloom(tortoise, color, length, petals)
```

The original function with eight petals has the turtle turn 1080/8 = 135 degrees so that it travels a total of 1080 degrees, a multiple of 360. When you generalize the number of petals, the sum of all of the angles that `tortoise` turns must

```
import turtle

def bloom(tortoise, color, length):
    tortoise.pencolor('red')
    tortoise.fillcolor(color)
    tortoise.begin_fill()
    for count in range(8):
        tortoise.forward(length)
        tortoise.left(135)
    tortoise.end_fill()

def stem(tortoise, length):
    tortoise.pencolor('green')
    tortoise.pensize(length / 20)
    tortoise.up()
    tortoise.forward(length / 2)
    tortoise.down()
    tortoise.right(90)
    tortoise.forward(length)

def flower(tortoise, color, length):
    bloom(tortoise, color, length)
    stem(tortoise, length)

george = turtle.Turtle()
george.hideturtle()
george.speed(6)

flower(george, 'yellow', 200)

screen = george.getscreen()
screen.exitonclick()
```

Figure 2.8 The final flower program.

still be a multiple of 360 (like 1080). Are there any values of `petals` for which your function does not work? Why?

2.3.4. Enhance the `stem` function so that it also draws a green leaf one third of the way up the stem, as shown to the right. One half of a pointed leaf can be drawn by repeatedly moving and turning the turtle small distances in a loop until it has turned a total of 90 degrees. The other half of the leaf can then be drawn by turning 90 degrees and repeating the same process.

2.3.5* Modify the `flower` function so that it creates a daffodil-like double bloom like the one to the right. The revised function will need two fill color parameters:

```
flower(tortoise, color1, color2, length)
```

It might help to know that the distance between any two opposite points of a bloom is about 1.08 times the segment length.

2.3.6. Write a function

```
growFlower(x, y, flowerColor, flowerLength)
```

that creates a turtle and then calls the `flower` function to draw a flower with that turtle at a particular (x,y) location. Test your new function by calling it from the flower program in Figure 2.8 in place of calling the `flower` function.

2.3.7. In this exercise, you will modify the `growFlower` function from the previous exercise so that it takes only `x` and `y` as parameters, and assigns `flowerColor` and `flowerLength` to random values in the body of the function. To do this, utilize two functions from the `random` module, which we will discuss more in Chapter 5. First, the `random.randrange` function returns a randomly chosen integer between its two arguments. To generate a random integer between 20 and 199 for the flower's size, call

```
flowerLength = random.randrange(20, 200)
```

Second, the `random.choice` function returns a randomly chosen item from a list. To generate a random color for the flower, call

```
flowerColor = random.choice(['yellow', 'pink', 'red', 'purple'])
```

Test your modified function as you did in the previous exercise. You will also need to import the `random` module at the top of your program.

2.3.8. In this exercise, you will implement the complete garden-drawing program from Figure 2.4 by modifying the program you wrote in the previous exercise so that it draws random flowers wherever you click in the drawing window. This will be accomplished by the following two methods of the `Screen` class:

```
screen.onclick(growFlower)
screen.mainloop()
```

Then the `mainloop` method repeatedly checks for mouse clicks and key presses, and calls designated functions when they happen. The `onclick` method indicates that `mainloop` should call `growFlower(x, y)` every time the mouse is clicked in the window at coordinates (x,y). To incorporate this functionality into your program, simply replace `screen.exitonclick()` in your program with these two statements.

2.3.9. Write a program that draws the word "CODE," as shown to the right. Use the `circle` method to draw the arcs of the "C" and "D." The `circle` method takes two arguments: the radius of the circle and the extent of the circle in degrees. For example, `george.circle(100, 180)` would draw half of a circle with radius 100. Making the extent negative draws the arc in the reverse direction.

2.3.10. Rewrite your program from Exercise 2.3.9 so that each letter is drawn by its own function. Then use your functions to draw "DECODE." (Call your "D" and "E" functions twice.)

2.3.11. Write a function

```
drawSquare(tortoise, width)
```

that uses the turtle named `tortoise` to draw a square with the given `width`. This function generalizes the code you wrote for Exercise 2.2.5 so that it can draw a square with any width. Use a `for` loop.

2.3.12. Write a function

```
drawRectangle(tortoise, length, width)
```

that uses the turtle named `tortoise` to draw a rectangle with the given `length` and `width`. This function generalizes the code you wrote for Exercise 2.2.6 so that it can draw a rectangle of any size. Use a `for` loop.

2.3.13. Write a function

```
drawPolygon(tortoise, sideLength, numSides)
```

that uses the turtle named `tortoise` to draw a regular polygon with the given number of sides and side length. This function is a generalization of your `drawSquare` function from Exercise 2.3.11. Use the value of `numSides` in your `for` loop and create a new variable for the turn angle that depends on `numSides`. The turtle will need to travel a total of 360 degrees over the course of the loop.

2.3.14. Write a function

```
drawCircle(tortoise, radius)
```

that calls your `drawPolygon` function from Exercise 2.3.13 to approximate a circle with the given radius.

2.3.15* Write a function

```
horizontalCircles(tortoise)
```

that draws ten non-overlapping circles, each with radius 50, that run horizontally across the graphics window. Use a `for` loop.

2.3.16. Write a function

```
diagonalCircles(tortoise)
```

that draws ten non-overlapping circles, each with radius 50, that run diagonally, from the top left to the bottom right, of the graphics window. Use a `for` loop.

2.3.17. Write a function

```
drawRow(tortoise)
```

that draws one row of an 8×8 red/black checkerboard. Use a `for` loop and the `drawSquare` function you wrote in Exercise 2.3.11.

2.3.18. Write a function

```
drawRow(tortoise, color1, color2)
```

that draws one row of an 8×8 checkerboard in which the colors of the squares alternate between `color1` and `color2`. The parameters `color1` and `color2` are both strings representing colors. For example, calling `drawRow(george, 'red', 'black')` should draw a row that alternates between red and black.

2.3.19. Write a function

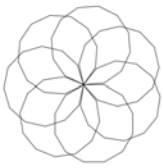
```
checkerBoard(tortoise)
```

that draws an 8×8 red/black checkerboard, using a `for` loop and the function you wrote in Exercise 2.3.18.

2.3.20. Interesting flower-like shapes can also be drawn by repeatedly drawing polygons that are rotated some number of degrees each time. Write a new function

```
polyFlower(tortoise, sideLength, numSides, numPolygons)
```

that calls the `drawPolygon` function from Exercise 2.3.13 to draw an interesting flower design. The function will repeatedly call `drawPolygon` a number of times equal to the parameter `numPolygons`, rotating the turtle each time to make a flower pattern. You will need to figure out the rotation angle based on the number of polygons drawn. For example, the image to the right was drawn by calling `drawFlower(george, 40, 12, 7)`.



2.3.21. Rewrite your program from Exercise 2.2.8 so that all of the drawing is done inside a function

```
plotSites(sites)
```

that takes the list of sites as a parameter. In other words, modify your program so that it looks like this:

```
import turtle

def plotSites(sites):
    drawing statements here...

sites = [(1.45, 7.31), (2.99, 7.55), (7.58, 6.29), (2.17, 4.71),
         (1.07, 5.56), ... ]

plotSites(sites)
```

2.3.22. Rewrite your program from Exercise 2.2.9 so that all of the drawing is done inside a function

```
plotPopulation(population)
```

that takes the population list as a parameter. In other words, modify your program so that it looks like this:

```
import turtle

def plotPopulation(population):
    drawing statements here...

population = [(1950, 2.557), (1951, 2.594), (1952, 2.636),
              (1953, 2.681), (1954, 2.73), (1955, 2.782), ... ]

plotPopulation(population)
```

2.3.23. Write a function

```
randomWalk(steps)
```

that generalizes your random walk code from Exercise 2.2.11 so that it draws a random walk for the given number of steps.

The following additional exercises ask you to write functions that do not involve turtle graphics. Test each one by calling it with both common and boundary case arguments, as described on page 38, and document your test cases. Use a trace table on at least one test case.

2.3.24. Write a function

```
basketball(fieldGoals, threePointers)
```

that prints your team's basketball score if the numbers of two pointers and three pointers are given in the parameters `fieldgoal` and `threePointers`.

2.3.25. Write a function

```
age(birthYear)
```

that prints a person's age when given his or her birth year as a parameter. You can assume that this function only works this year and that the person has not had his or her birthday yet this year.

2.3.26. Write a function

```
cheer(teamName)
```

that takes as a parameter a team name and prints "Go" followed by the team name. For example, if the function is called as `cheer('Buzzards')`, it should print the string `'Go Buzzards'` to the screen.

2.3.27. Write a function

```
sum(number1, number2)
```

that prints the sum of `number1` and `number2` to the screen.

2.3.28. Write a function

```
printTwice(word)
```

that prints its parameter twice on two separate lines.

2.3.29. Write a function

```
printMyName()
```

that uses a `for` loop to print your name 100 times.

2.4 PROGRAMMING IN STYLE

Programming style and writing style share many of the same concerns. When we write an essay, we want the reader to clearly understand our thesis and the arguments that support it. We want it to be clear and concise, and have a logical flow from beginning to end. Similarly, when we write a program, we want to help collaborators understand our program's goal, how it accomplishes that goal, and how it flows from beginning to end. Even if you are the only one to ever read your program, good style will pay dividends both while you are working through the solution, and in the future when you try to reacquaint yourself with your work. We can accomplish these goals by organizing our programs neatly and logically, using descriptive variable and

function names, writing programs that accomplishes their goal in a non-obfuscated manner, and documenting our intentions within the program.

Program structure

Let's return to the program that we wrote in the previous section (Figure 2.8), and reorganize it a bit to reflect better programming habits. As shown in Figure 2.9, every program should begin with documentation that identifies the program's author and its purpose. This type of documentation, which starts and ends with three double quotes (`"""`), is called a ***docstring***; we will look more closely at docstrings and other types of documentation shortly.

We follow this with our `import` statements. Putting these at the top of our program both makes our program neater and ensures that the imported modules are available anywhere later on.

Next, we define all of our functions. Because programs are read by the interpreter from top to bottom, you need to define your functions above where you call them. For example, if we tried to call the `bloom` function at the very top of the program, before it was defined, we would generate an error message.

At the end of the flower-drawing program in Figure 2.8, there are six statements at the outermost indentation level. The first and fifth of these statements define global variable names that are visible and potentially modifiable anywhere in the program. When the value assigned to a global variable is modified in a function, it is called a ***side effect***. In large programs, where the values of global variables can be potentially modified in countless different places, errors in their use become nearly impossible to find. For this reason, we should get into the habit of never using them, unless there is a *very* good reason, and these are pretty hard to come by. See Tangent 2.2 for more information on how global names are handled in Python.

To prevent the use of global variables, and to make programs more readable, we will move statements at the global level of our programs into a function named `main`, and then call `main` as the last statement in the program, as shown at the end of the program in Figure 2.9. With this change, the call to the `main` function is where the action begins in this program. (Remember that the function definitions above only define functions; they do not execute them.) The `main` function sets up a turtle, then calls our `flower` function, which then calls the `bloom` and `stem` functions. Getting used to this style of programming has an additional benefit: it is very similar to the style of other common programming languages (e.g., C, C++, Java) so, if you go on to use one of these in the future, it should seem relatively familiar.

The functions in a program are generally determined by how the problem was decomposed during the top-down design process. Even so, identifying functions can be as much an art as a science, so here are a few guidelines to keep in mind:

1. A function should accomplish something relatively small, and make sense standing on its own.

program docstring

```
"""
Purpose: Draw a flower
Author: Ima Student
Date: September 15, 2020
CS 111, Fall 2020
"""
```

import statements

```
import turtle
```

function definitions

```
def bloom(tortoise, color, length):
    """Draws a geometric flower bloom.

    Parameters:
        tortoise: a Turtle object with which to draw the bloom.
        color: a color string to use to fill the bloom.
        length: the length of each segment of the bloom.

    Return value:
        None
    """

    tortoise.pencolor('red')     # set tortoise's pen color to red
    tortoise.fillcolor(color)    # and fill color to fcolor
    tortoise.begin_fill()
    for segment in range(8):     # draw a filled 8-sided
        tortoise.forward(length) #   geometric flower bloom
        tortoise.left(135)
    tortoise.end_fill()

# other functions omitted...
```

main function

```
def main():
    """Draws a yellow flower with segment length 200, and
       waits for a mouse click to exit.
    """

    george = turtle.Turtle()
    george.hideturtle()
    george.speed(6)

    flower(george, 'yellow', 200)

    screen = george.getscreen()
    screen.exitonclick()
```

main function call

```
main()
```

Figure 2.9 An overview of a program's structure.

2. Functions should be written for subproblems that are called upon frequently, perhaps with different arguments. If you find yourself duplicating some part of a program, write a function for it instead.
3. A function should generally fit on a page or, in many cases, less.
4. The `main` function should be short, generally serving only to set up the program and call other functions that carry out the work.

Documentation

Python program documentation comes in two flavors: docstrings and ***comments***. A docstring is meant to articulate everything that someone needs to know to use a program or module, or to call a function. Comments, on the other hand, are used to

Tangent 2.2: Global variables

The Python interpreter handles global names inside functions differently, depending on whether the name's value is being read or the name is being assigned a value. When the Python interpreter encounters a name that needs to be evaluated (e.g., on the righthand side of an assignment statement), it first looks to see if this name is defined inside the scope of this function. If it is, the name in the local scope is used. Otherwise, the interpreter successively looks at outer scopes until the name is found. If it reaches the global scope and the name is still not found, we see a "name error."

On the other hand, if we assign a value to a name, that name is always considered to be local, unless we have stated otherwise by using a `global` statement. For example, consider the following program:

```
spam = 13

def func1():
    spam = 100

def func2():
    global spam
    spam = 200

func1()
print(spam)
func2()
print(spam)
```

The first `print` will display 13 because the assignment statement that is executed in `func1` defines a new local variable; it does not modify the global variable with the same name. But the second `print` will display 200 because the `global` statement in `func2` indicates that `spam` should refer to the global variable with that name, causing the subsequent assignment statement to change the value assigned to the global variable. This convention prevents accidental side effects because it forces the programmer to explicitly decide to modify a global variable. In any case, using `global` is strongly discouraged.

document individual program statements or groups of statements. In other words, a docstring explains *what* a program or function does, while comments explain *how* it works; a docstring describes an abstraction while comments describe what happens inside the black box. The Python interpreter ignores both docstrings and comments while it is executing a program; both are intended for human eyes only.

Docstrings

A docstring is enclosed in a matching pair of triple double quotes (`"""`), and may occupy several lines. We use a docstring at the beginning of every program to identify

the program's author and its purpose, as shown at the top of Figure 2.9.[3] We also use a docstring to document each function that we write, to ensure that the reader understands what it does. A function docstring should articulate everything that someone needs to know to call the function: the overall purpose of the function, and descriptions of the function's parameters and return value.

The beginning of a function's docstring is indented on the line immediately following the `def` statement. Programmers prefer a variety of different styles for docstrings; we will use one that closely resembles the style in Google's official Python style guide. Docstrings for the three functions from Figure 2.8 are shown below. (The bodies of the functions are omitted.)

```
def bloom(tortoise, color, length):
    """Draws a geometric flower bloom.

    Parameters:
        tortoise: a Turtle object with which to draw the bloom
        color:    a color string to use to fill the bloom
        length:   the length of each segment of the bloom

    Return value:
        None
    """

def stem(tortoise, length):
    """Draws a flower stem.

    Parameters:
        tortoise: a Turtle object, initially at the bloom starting
                  position
        length:   the length of the stem and each segment of the bloom

    Return value:
        None
    """

def flower(tortoise, color, length):
    """Draws a flower.

    Parameters:
        tortoise: a Turtle object with which to draw the flower
        color:    a color string to use to fill the bloom
        length:   the length of each segment of the bloom

    Return value:
        None
    """
```

[3] Your instructor may require a different format, so be sure to ask.

In the first line of the docstring, we succinctly explain what the function does. This is followed by a parameter section that lists each parameter with its intended purpose and the class to which it should belong. If there are any assumptions made about the value of the parameter, these should be stated also. For example, the turtle parameter of the `stem` function is assumed to start at the origin of the bloom. Finally, we describe the return value of the function. We did not have these functions return anything, so they return `None`. We will look at how to write functions that return values in Section 2.5.

Another advantage of writing docstrings is that Python can automatically produce documentation from them, in response to calling the `help` function. For example, try this short example in the Python shell:

```
>>> def printName(first, last):
        """Prints a first and last name.

        Parameters:
            first: a first name
            last:  a last name

        Return value:
            None
        """

        print(first + ' ' + last)
>>> help(printName)
Help on function printName in module __main__:

printName(first, last)
    Prints a first and last name.

    Parameters:
        first: a first name
        last:  a last name

    Return value:
        None
```

You can also use `help` with modules and built-in functions. For example, try this:

```
>>> import turtle
>>> help(turtle.color)
```

Comments

A comment is anything between a hash symbol (`#`) and the end of the line. As with docstrings, the Python interpreter ignores comments. Comments should generally be neatly lined up to the right of the statements they document. However, there are times when a longer comment is needed to explain a complicated section. In this case, you might want to precede that section with a comment on one or more lines by itself.

There is a fine line between under-commenting and over-commenting. As a general rule, you want to supply high-level descriptions of what your code intends to do. You do *not* want to literally repeat what each individual line does, as this is not at all helpful to someone reading your code. Doing so tends to clutter it up and make it *harder* to read! Here are examples of good comments for the body of the `bloom` function.

```
tortoise.pencolor('red')       # set tortoise's pen color
tortoise.fillcolor(color)      #   and fill color
tortoise.begin_fill()
for count in range(8):         # draw a filled 8-sided
    tortoise.forward(length)   #   geometric flower bloom
    tortoise.left(135)
tortoise.end_fill()
```

Notice that the five lines that draw the bloom are commented together, just to note the programmer's intention. In contrast, the following comments illustrate what *not* to do. The following comments are both hard to read and uninformative.

```
tortoise.pencolor('red')  # set tortoise's pen color to red
tortoise.fillcolor(color)  # set tortoise's fill color to color
tortoise.begin_fill()  # begin to fill a shape
for count in range(8):  # for count = 0, 1, 2, ..., 7
    tortoise.forward(length)  # move tortoise forward length
    tortoise.left(135)  # turn tortoise left 135 degrees
tortoise.end_fill()  # stop filling the shape
```

Notice that these comments never actually explain the purpose of the for loop; they just repeat each line. Instead, as above, you want to step back and explain the purpose of the code and, only if it is not obvious, how it is accomplished. We leave the task of commenting the other functions in this program as an exercise.

Self-documenting code

As we discussed in Section 1.3, using descriptive variable names is a very important step in making your program's intentions clear. The variable names in the flower program are already in good shape, so let's look at a different example. Consider the following statements.

```
x = 462
y = (3.95 - 1.85) * x - 140
```

Without any context, it is impossible to infer what this is supposed to represent. However, if we rename the two variables, as follows, the meaning becomes clearer.

```
cupsSold = 462
profit = (3.95 - 1.85) * cupsSold - 140
```

Now it is clear that this code is computing the profit generated from selling cups of something. But the meaning of the numbers is still a mystery. These are examples of ***magic numbers***, so-called in programming parlance because they seem to appear out of nowhere. There are at least two reasons to avoid magic numbers. First, they make your code less readable and obscure its meaning. Second, they make it more

difficult and error-prone to change your code, especially if you use the same value multiple times. By assigning these numbers to descriptive variable names, the code becomes even clearer.

```
cupsSold = 462
pricePerCup = 3.95
costPerCup = 1.85
fixedCost = 140
profit = (pricePerCup - costPerCup) * cupsSold - fixedCost
```

We now have ***self-documenting code***. Since we have named all of our variables and values with descriptive names, just reading the code is enough to deduce its intention. These same rules, of course, apply to function names and parameters. By naming our functions with descriptive names, we make their purposes clearer and we contribute to the readability of the functions from which we call them. This practice will continue to be demonstrated in the coming chapters.

In this book, we use a naming convention that is sometimes called `camelCase`, in which the first letter is in lowercase and then the first letters of subsequent words are capitalized. But other programmers prefer different styles. For example, some programmers prefer `snake_case`, in which an underscore character is placed between words (`cupsSold` would be `cups_sold`). Unless you are working in an environment with a specific mandated style, the choice is yours, as long as it results in self-documenting code.

Exercises

2.4.1* Incorporate all the changes we discussed in this section into your flower-drawing program, and finish commenting the bodies of the remaining functions.

2.4.2. Reorganize the earthquake plotting program from page 63 so that it follows all of the style guidelines from this section, and the actual drawing is encapsulated in a function `plotQuakes(tortoise, earthquakes)`. A `main` function should create a turtle, assign the list of earthquakes to a variable, and then call your function with these two arguments.

2.4.3* Rewrite this simple program so that it adheres to the guidelines in this section. All of the drawing should happen in a new function that takes the name of a turtle as its parameter. The `main` function should create a turtle and pass it into the drawing function. Be sure to include docstrings and comments.

```
import turtle
beth = turtle.Turtle()
beth.hideturtle()
beth.speed(9)
beth.fillcolor('blue')
beth.begin_fill()
beth.pencolor('red')
for count in range(8):
    beth.circle(75)
    beth.left(45)
    beth.forward(10 * 1.414)   # 10 * sqrt(2)
beth.end_fill()
```

2.4.4. Run the following program to see what it does and then edit it to make it more understandable. Give all of the variables more descriptive names and add appropriate docstrings and comments.

```
import turtle
import math     # math module (more in the next chapter)

def doSomething(z):
    a = turtle.Turtle()
    b = turtle.Turtle()
    c = a.getscreen()
    c.setworldcoordinates(-z - 1, -z - 1, z + 1, z + 1)
    a.hideturtle()
    b.hideturtle()

    a.up()
    b.up()
    a.goto(-z, 0)
    b.goto(-z, 0)
    a.down()
    b.down()

    for d in range(-z, z + 1):
        a.goto(d, math.sqrt(z ** 2 - d ** 2))  # sqrt is square root
        b.goto(d, -math.sqrt(z ** 2  - d ** 2))

def main():
    doSomething(100)

main()
```

2.4.5* Write a program that prompts for a person's age and then prints the equivalent number of days. All of the statements should be in a `main` function.

2.4.6. Write a program that prompts for a person's favorite color and the last thing they ate. Then print the concatenation of these as their rock band name. All of the statements should be in a `main` function. For example:

```
Your favorite color? pink
Your last meal? burrito
Your band name is The pink burritos!
```

2.4.7. Write a function that implements your Mad Lib from Exercise 1.3.22, and then write a complete program (with `main` function) that calls it. Your Mad Lib function should take the words needed to fill in the blanks as parameters. Your `main` function should get these values with calls to the `input` function, and then pass them to your function. Include docstrings and comments in your program. For example, here is a new version of the example in Exercise 1.3.22 (without docstrings or comments).

```
def party(adj1, noun1, noun2, adj2, noun3):
    print('How to Throw a Party')
    print()
    print('If you are looking for a/an', adj1, 'way to')
    print('celebrate your love of', noun1 + ', how about a')
    print(noun2 + '-themed costume party?  Start by')
    print('sending invitations encoded in', adj2, 'format')
    print('giving directions to the location of your', noun3 + '.')

def main():
    firstAdj = input('Adjective: ')
    firstNoun = input('Noun: ')
    secondNoun = input('Noun: ')
    secondAdj = input('Adjective: ')
    thirdNoun = input('Noun: ')
    party(firstAdj, firstNoun, secondNoun, secondAdj, thirdNoun)

main()
```

2.4.8. Study the following program (also available on the book website), and then reorganize it with a `main` function that calls one or more other functions. Your `main` function should only create a turtle and call your functions. Document your program with appropriate docstrings and comments.

```
import turtle
george = turtle.Turtle()
george.setposition(0, 100)
george.pencolor('red')
george.fillcolor('red')
george.begin_fill()
george.circle(-100, 180)
george.right(90)
george.forward(200)
george.end_fill()
george.up()
george.right(90)
george.forward(25)
george.right(90)
george.forward(50)
george.left(90)
george.down()
george.pencolor('white')
george.fillcolor('white')
george.begin_fill()
george.circle(-50, 180)
george.right(90)
george.forward(100)
george.end_fill()
```

2.4.9. The following program (also available on the book website) draws a truck. Edit it so that it conforms to all of the guidelines discussed in this section. Include all code in appropriate functions and replace duplicate code with appropriate function calls.

```
import turtle

truck = turtle.Turtle()
truck.speed(5)
truck.hideturtle()

truck.fillcolor('red')
truck.begin_fill()
truck.forward(300)
truck.left(90)
truck.forward(75)
truck.left(45)
truck.forward(25)
truck.left(45)
truck.forward(100)
truck.right(45)
truck.forward(100)
truck.left(45)
truck.forward(75)
truck.left(90)
truck.forward(70.71)
truck.right(90)
truck.forward(200)
truck.left(90)
truck.forward(92.677)
truck.left(90)
truck.forward(167.677)
truck.end_fill()

truck.up()
truck.forward(220)
truck.right(90)
truck.forward(50)
truck.left(90)
truck.down()
truck.fillcolor('black')
truck.begin_fill()
truck.circle(50)
truck.end_fill()
truck.up()
truck.right(90)
truck.backward(25)
truck.left(90)
truck.down()
truck.fillcolor('lightgray')
truck.begin_fill()
truck.circle(25)
truck.end_fill()

truck.up()
truck.right(90)
truck.forward(25)
truck.left(90)
truck.backward(300)
truck.down()
truck.fillcolor('black')
truck.begin_fill()
truck.circle(50)
truck.end_fill()

truck.up()
truck.right(90)
truck.backward(25)
truck.left(90)
truck.down()
truck.fillcolor('lightgray')
truck.begin_fill()
truck.circle(25)
truck.end_fill()
```

2.5 A RETURN TO FUNCTIONS

We previously described a function as a computation that takes one or more inputs, called *parameters*, and produces an output, called a *return value*. But, up until now, very few of the functions we have used, and none of the functions we have written, have had return values. In this section, we will remedy that.

The math module

The `math` module contains a rich set of functions that return useful mathematical quantities. For example, to take the square root of 5, we can use `math.sqrt`:

```
>>> import math
>>> result = math.sqrt(5)
>>> result
2.23606797749979
```

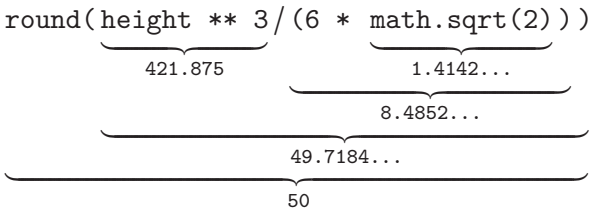
Function calls with return values can also be used in longer expressions, and as arguments of other functions. In this case, it is useful to think about a function call as equivalent to the value that it returns. For example, we can use the `math.sqrt` function in the computation of the volume of a tetrahedron with edge length $h = 7.5$, using the formula $V = h^3/(6\sqrt{2})$.

```
>>> height = 7.5
>>> volume = height ** 3 / (6 * math.sqrt(2))
>>> volume
49.71844555217912
```

In the parentheses, the value of `math.sqrt(2)` is computed first, and then multiplied by 6. Finally, `height ** 3` is divided by this result, and the answer is assigned to `volume`. If we wanted the rounded volume, we could use the entire volume computation as the argument to the `round` function:

```
>>> volume = round(height ** 3 / (6 * math.sqrt(2)))
>>> volume
50
```

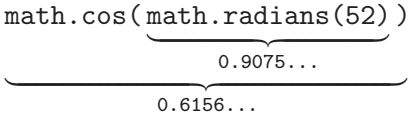
We illustrate below the complete sequence of events in this evaluation:



Now suppose we wanted to find the cosine of a 52° angle. We can use the `math.cos` function to compute the cosine, but the Python trigonometric functions expect their arguments to be in radians instead of degrees. (360 degrees is equivalent to 2π radians.) Fortunately, the `math` module also provides a function named `radians` that converts degrees to radians. So we can find the cosine of a 52° angle like this:

```
>>> math.cos(math.radians(52))
0.6156614753256583
```

The function call `math.radians(52)` is evaluated first, giving the equivalent of 52° in radians, and this result is used as the argument to the `math.cos` function:



Other commonly used functions from the `math` module are listed in Appendix A.1. The `math` module also contains two commonly used constants: `pi` and `e`. Our sphere volume computation earlier would have been more accurately computed with:

```
>>> radius = 20
>>> volume = (4 / 3) * math.pi * (radius ** 3)
>>> volume
33510.32163829113
```

Notice that, since `pi` and `e` are variable names, not functions, there are no parentheses after their names.

Writing functions with return values

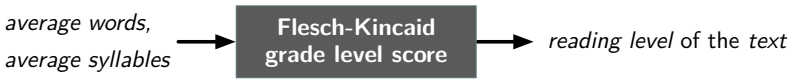
When we computed Flesch-Kincaid grade levels back in Section 1.3, they looked like this:

```
>>> averageWords = 16
>>> averageSyllables = 1.78
>>> readingLevel = 0.39 * averageWords + 11.8 * averageSyllables - 15.59
>>> print(readingLevel)
11.654
```

The problem with this approach is that, if we want the reading level of a different text, we need to type the whole thing again. For example,

```
>>> averageWords = 4.8
>>> averageSyllables = 1.9
>>> readingLevel = 0.39 * averageWords + 11.8 * averageSyllables - 15.59
>>> print(readingLevel)
8.702000000000002
```

This is obviously tedious and error-prone, analogous to building a new microwave oven from scratch every time we want to pop a bag of popcorn. Instead, we want to define a function that takes the average words per sentence and the average syllables per word as inputs and returns the reading level as output. Once we have this function in hand, we can get the reading levels of as many books as we want just by passing in different arguments. Back in Chapter 1, we visualized the problem as a black box like this:



And we wrote the algorithm in pseudocode like this:

Algorithm FLESCH KINCAID

Input: *average words, average syllables*

1 | *reading level* ← 0.39 × *average words* + 11.8 × *average syllables* − 15.59

Output: *reading level*

In Python, these translate into the following function definition.

```
def fleschKincaid(averageWords, averageSyllables):
    """Computes the reading level of a text using the Flesch-Kincaid
       reading level formula.

    Parameters:
        averageWords:     average number of words per sentence in a text
        averageSyllables: average number of syllables per word in a text

    Return value: the Flesch-Kincaid reading level score
    """

    return 0.39 * averageWords + 11.8 * averageSyllables - 15.59
```

The `return` statement defines the output of the function. Remember that the function definition by itself does not compute anything. We must call the function for it to be executed. For example, to get the two reading levels above, we can call the function twice like this:

```
def main():
    readingLevel1 = fleschKincaid(16, 1.78)
    readingLevel2 = fleschKincaid(4.8, 1.9)

    print('The reading level of book 1 is ' + str(readingLevel1) + '.')
    print('The reading level of book 2 is ' + str(readingLevel2) + '.')
    print('The difference in reading level is ' +
                              str(readingLevel1 - readingLevel2) + '.')

main()
```

When each assignment statement is executed, the righthand side calls the function `fleschKincaid` with two arguments. Then the function `fleschKincaid` is executed with the two arguments assigned to its two parameters. Next, the value after the `return` statement is computed and returned by the function. In `main`, this return value is assigned to the variable on the left of the assignment statement. So when this program is run, it prints:

```
The reading level of book 1 is 11.654.
The reading level of book 2 is 8.702000000000002.
The difference in reading level is 2.951999999999998.
```

The return value becomes the value associated with the function call itself. For example, the first `print` statement could be changed to

```
print('The reading level of book 1 is ' + str(fleschKincaid(6, 2.2)) + '.')
```

In addition to defining a function's return value, the `return` statement *also* causes the function to end and return this value back to the function call. So the `return` statement actually does two things:

1. defines the function's return value, *and*
2. causes the function to end.

This second point is important to remember because it means that any statements we add to a function after the `return` statement will never be executed.

Reflection 2.11 *Add the statement*

```
print('This will never, ever be printed.')
```

to the `fleschKincaid` *function after the* `return` *statement. What does it do?*

Functions can have many statements in them before the `return` statement. The following function gets characteristics about a text by prompting for them, then calls our new `fleschKincaid` function and returns the result.

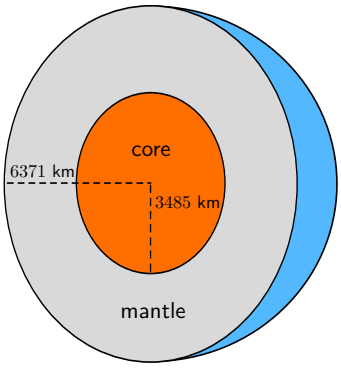
```
def fleschKincaid2():
    """Prompt for characteristics about a text and then return the
       text's reading level according to the Flesch-Kincaid formula.

    Parameters: none

    Return value: the Flesch-Kincaid reading level score
    """

    averageWords = float(input('Average words per sentence: '))
    averageSyllables = float(input('Average syllables per word: '))

    readingLevel = fleschKincaid(averageWords, averageSyllables)

    return readingLevel
```

By defining functions that return values, we can also add to the existing palette of mathematical functions supplied by Python. For example, our familiar sphere volume computation looks like this as a Python function:

```
import math

def volumeSphere(radius):
    """Computes the volume of a sphere.

    Parameter:
        radius: radius of the sphere

    Return value: volume of a sphere with the given radius
    """

    return (4 / 3) * math.pi * (radius ** 3)
```

Now suppose we want to approximate the volume of the earth's mantle, which is the layer between the earth's core and its crust. This is the same as computing the

difference of two volumes: the volume of the earth and the volume of the earth's core, as illustrated below.




```
earthRadius = 6371 # km
coreRadius = 3485  # km
mantleVolume = volumeSphere(earthRadius) - volumeSphere(coreRadius)
print("The volume of the mantle is " + str(mantleVolume) + ' cubic km.')
```

Notice how we used two function calls in an arithmetic expression, exactly like we previously used the `int` and `math.sqrt` functions. This expression is evaluated from the inside out, just as one would expect:

```
mantleVolume = volumeSphere(earthRadius) - volumeSphere(coreRadius)
                            6371                 3485
               1083206916845.7535   177295191309.51987
               905911725536.2336
```

Return vs. print

A common beginner's mistake is to forget the `return` statement or end a function with a `print` instead of a `return`. For example, suppose we replaced the `return` with `print` in the `volumeSphere` function:

```
def fleschKincaid(averageWords, averageSyllables):
    """ (docstring omitted) """

    print(0.39 * averageWords + 11.8 * averageSyllables - 15.59) # WRONG!
```

Reflection 2.12 *Make this modification to your* `fleschKincaid` *function and then run your program again. Did you get the correct answer?*

When you run your program now, you will see something puzzling:

```
11.654
8.702000000000002
The reading level of book 1 is None.
The reading level of book 2 is None.
TypeError: unsupported operand type(s) for -: 'NoneType' and 'NoneType'
```

The first two lines where the reading levels are printed are coming from the `print`

in the `fleschKincaid` function. Because there is no `return` in the function, the return value is `None`, which is assigned to `readingLevel1` and `readingLevel2`, and printed in the third and fourth lines. The fifth line is the error generated by trying to compute `readingLevel1 - readingLevel2`, which is `None - None`, a nonsensical operation.

Reflection 2.13 *A similar problem will arise if you replace the last statement in* `fleschKincaid` *function with*

```
readingLevel = 0.39 * averageWords + 11.8 * averageSyllables - 15.59
```

and omit a `return` *statement. Try it. What happens and why?*

Before continuing, be sure to fix your function so that it has a proper `return` statement.

Exercises

The following exercises ask you to write functions that `return` *(not* `print`*) values. When a program is called for, be sure to follow the guidelines in the previous section. Test each function with both common and boundary case arguments, as described on page 38, and document these test cases.*

2.5.1* The geometric mean of two numbers is the square root of their product. Write a function

```
geometricMean(value1, value2)
```

that returns the geometric mean of the two values. Use your function to compute the geometric mean of 18 and 31.

2.5.2. If you have P (short for principal) dollars in a savings account that will pay interest rate r, compounded at a frequency of n times per year, then after t years, you will have

$$P\left(1+\frac{r}{n}\right)^{nt}$$

dollars in your account. If the interest were compounded continuously (i.e., with n approaching infinity), you would instead have

$$Pe^{rt}$$

dollars after t years, where e is Euler's number, the base of the natural logarithm. Write a function

```
compoundDiff(principal, rate, frequency, years)
```

that returns the difference in your savings between compounding at the given `frequency` and continuous compounding. (Use the `math.exp` function.)

Suppose you have P = \$10,000 in an account paying 1% interest (r = 0.01), compounding monthly. Use your function to determine how much more money will you have after t = 10 years if the interest were compounded continuously.

2.5.3. Write a program that prompts for a principal, rate, compounding frequency, and number of years, and then uses your function from Exercise 2.5.2 to display how much more money will you have if the interest were compounded continuously.

2.5.4* Write a function

```
quadratic(a, b, c)
```

that uses the quadratic formula to return the two solutions to the equation $ax^2 + bx + c = 0$. Your function can `return` two values like this:

```
return x1, x2
```

Show how to use your function to find the solutions to $3x^2 + 4x - 5 = 0$.

2.5.5. Write a program that prompts for values of a, b, and c using the `input` function, calls your function from Exercise 2.5.4 to find the solutions to the quadratic equation $ax^2 + bx + c = 0$, and then prints the results.

2.5.6* Suppose we have two points (x_1, y_1) and (x_2, y_2). The distance between them is equal to

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

Write a function

```
distance(x1, y1, x2, y2)
```

that returns the distance between points `(x1,y1)` and `(x2,y2)`.

2.5.7. A parallelepiped is a three-dimensional box in which the six sides are parallelograms. The volume of a parallelepiped is

$$V = abc\sqrt{1 + 2\cos(x)\cos(y)\cos(z) - \cos(x)^2 - \cos(y)^2 - \cos(z)^2}$$

where a, b, and c are the edge lengths, and x, y, and z are the angles between the edges, in radians. Write a function

```
ppdVolume(a, b, c, x, y, z)
```

that returns the volume of a parallelepiped with the given dimensions.

2.5.8. Repeat the previous exercise, but now assume that the angles passed into the function are in degrees.

2.5.9. Write a function

```
total(number1, number2)
```

that returns the sum of `number1` and `number2`. Also write a complete program (with a `main` function) that gets these two values using the `input` function, passes them to your `total` function, and then prints the value returned by the `total` function.

2.5.10* Write a function

```
power(base, exponent)
```

that returns the value $\texttt{base}^{\texttt{exponent}}$. Also write a complete program (with a `main` function) that gets these two values using the `input` function, passes them to your `power` function, and then prints the returned value of $\texttt{base}^{\texttt{exponent}}$.

2.5.11* Write a function

```
football(touchdowns, fieldGoals, safeties)
```

that returns your team's football score if the number of touchdowns (worth 7 points), field goals (worth 3 points), and safeties (worth 2 points) are passed as parameters. Then write a complete program (with `main` function) that gets these three values using the `input` function, passes them to your `football` function, and then prints the score.

2.5.12. The ideal gas law states that $PV = nRT$ where

- P = pressure in atmospheres (atm)
- V = volume in liters (L)
- n = number of moles (mol) of gas
- R = gas constant = 0.08 L atm / mol K
- T = absolute temperature of the gas in Kelvin (K)

Write a function

```
moles(V, P, T)
```

that returns the number of moles of an ideal gas in V liters contained at pressure P and T degrees Celsius. (Be sure to convert Celsius to Kelvin in your function.) Also write a complete program (with a `main` function) that gets these three values using the `input` function, passes them to your `moles` function, and then prints the number of moles of ideal gas.

2.5.13. Suppose we have two containers of an ideal gas. The first contains 10 L of gas at 1.5 atm and 20 degrees Celsius. The second contains 25 L of gas at 2 atm and 30 degrees Celsius. Show how to use two calls to your function in the previous exercise to compute the total number of moles of ideal gas in the two containers.

Now replace the `return` statement in your `moles` function with a call to `print` instead. (So your function does not contain a `return` statement.) Can you still compute the total number of moles in the same way? If so, show how. If not, explain why not.

2.5.14* Most of the world is highly dependent upon groundwater for survival. Therefore, it is important to be able to monitor groundwater flow to understand potential contamination threats. Darcy's law states that the flow of a liquid (e.g., water) through a porous medium (e.g., sand, gravel) depends upon the capacity of the medium to carry the flow and the gradient of the flow:

$$Q = K\frac{dh}{dl}$$

where

- K is the hydraulic conductivity of the medium, the rate at which the liquid can move through it, measured in area/time
- dh/dl is the hydraulic gradient
- dh is the drop in elevation (negative for flow down)
- dl is the horizontal distance of the flow

Write a function

```
darcy(K, dh, dl)
```

that computes the flow with the given parameters.

Use your function to compute the amount of groundwater flow inside a hill with hydraulic conductivity of 130 m^2/day, and a 50 m drop in elevation over a distance of 1 km.

2.5.15. A person's Body Mass Index (BMI) is calculated by the following formula:

$$\text{BMI} = \frac{w}{h^2} \cdot 703$$

where w is the person's weight in pounds and h is the person's height in inches. Write a function

```
bmi(weight, height)
```

that uses this formula to return the corresponding BMI.

2.5.16* When you (or your parents) rip songs from a CD, the digital file is created by sampling the sound at some rate. Common rates are 128 kbps (128×2^{10} bits per second), 192 kbps, and 256 kbps. Write a function

```
songs(capacity, bitrate)
```

that returns the number of 4-minute songs someone can fit locally on his or her music player. The function's two parameters are the capacity of the music player in gigabytes (GB) and the sampling rate in kbps. A gigabyte is 2^{30} bytes and a byte contains 8 bits. Also write a complete program (with a `main` function) that gets these two values using the `input` function, passes them to your `songs` function, and then prints the number of songs.

2.5.17. The speed of a computer is often (simplistically) expressed in gigahertz (GHz), the number of billions of times the computer's internal clock "ticks" per second. For example, a 2 GHz computer has a clock that "ticks" 2 billion times per second. Suppose that a single computer instruction requires 3 "ticks" to execute. Write a function

```
time(instructions, gigahertz)
```

that returns the time in seconds required to execute the given number of `instructions` on a computer with clock rate `gigahertz`. For example, `time(10 ** 9, 3)` should return `1` (second).

2.5.18* Exercise 1.3.8 asked how to swap the values in two variables. Can we write a function to swap the values of two parameters? In other words, can we write a function

```
swap(a, b)
```

and call it like

```
x = 10
y = 1
swap(x, y)
```

so that after the function returns, `x` has the value 1 and `y` has the value 10? (The function should not return anything.) If so, write it. If not, explain why not.

2.5.19. Given an integer course grade from 0 to 99, we convert it to the equivalent grade point according to the following scale: 90–99: 4, 80–89: 3, 70–79: 2, 60–69: 1, < 60: 0. Write a function

```
gradePoint(score)
```

that returns the grade point (i.e., GPA) equivalent to the given score.

2.5.20. The function `time.time()` (in the `time` module) returns the current time in seconds since January 1, 1970. Write a function

```
year()
```

that uses this function to return the current year as an integer value.

2.5.21* Write a function

```
twice(text)
```

that uses the string concatenation operator `*` to return the string `text` repeated twice, with a space in between. For example, `twice('bah')` should return the string `'bah bah'`.

2.5.22. Write a function

```
repeat(text, n)
```

that returns a string that is `n` copies of the string `text`. For example, `repeat('AB', 3)` should return the string `'ABABAB'`.

2.6 SCOPE AND NAMESPACES

We have been using local variables inside functions for a few sections now, relying on somewhat informal explanations for how they work. In this section, we will look more formally at scoping rules for variables so that you better understand how to use them and can hopefully prevent difficult-to-find errors in the future. As an example, let's consider the wind chill computation from Exercise 1.3.14, implemented as a function that is called from a `main` function.

```
def windChill(temperature, windSpeed):
    """Gives the North American metric wind chill equivalent
       for the given temperature and wind speed.

    Parameters:
        temperature: temperature in degrees Celsius
        windSpeed:   wind speed at 10m in km/h

    Return value:
        equivalent wind chill in degrees Celsius, rounded to
        the nearest integer
    """

    chill = 13.12 + 0.6215 * temperature \
                  + (0.3965 * temperature - 11.37) * windSpeed ** 0.16
    temperature = round(chill)
    return temperature

def main():
    temp = -3
    wind = 13
    chilly = windChill(temp, wind)
    print('The wind chill is ' + str(chilly) + ' degrees Celsius.')

main()
```

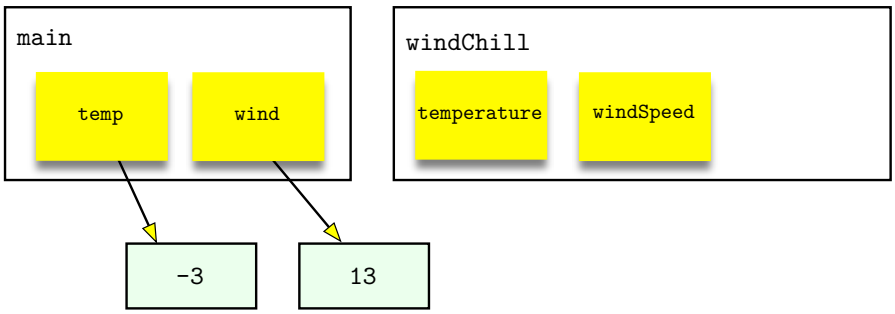
(The "backslash" (\) character above is the *line continuation character*. It indicates that the line that it ends is continued on the next line. This is sometimes handy for splitting very long lines of code.) Notice that we have introduced a variable inside the `windChill` function named `chill` to break up the computation a bit. Because we created `chill` inside the function `windChill`, its ***scope*** is local to the function. If we tried to refer to `chill` anywhere outside of the function `windChill` (e.g., in the `main` function), we would get the following error:

```
NameError: name 'chill' is not defined
```

Because `chill` has a local scope, it is called a ***local variable***. The parameters `temperature` and `windSpeed` are also local variables and have the same local scope as `chill`.

Local namespaces

Let's look more closely at how local variable and parameter names are managed in Python. In this program, just after we call the `windChill` function, but just before the values of the arguments `temp` and `wind` are assigned to the parameters `temperature` and `windSpeed`, we can visualize the situation like this:



The box around `temp` and `wind` represents the scope of the `main` function, and the box around `temperature` and `windSpeed` represents the scope of the `windChill` function. In each case, the scope defines what names have been defined, or have meaning, in that function. In the picture, we are using arrows instead of affixing the "Sticky notes" directly to the values to make clear that the names, not the values, reside in their respective scopes. The names are references to the memory cells in which their values reside.

The scope corresponding to a function in Python is managed with a ***namespace***. A namespace of a function is simply a list of names that are defined in that function, together with references to their values. We can view the namespace of a particular function by calling the `locals` function from within it. For example, insert the following statement into the `main` function, just before the call to `windChill`:

```
print('Local namespace in main before windChill is\n\t', locals())
```

(The `\n\t` represents a newline and tab character.) When we run the program, we will see

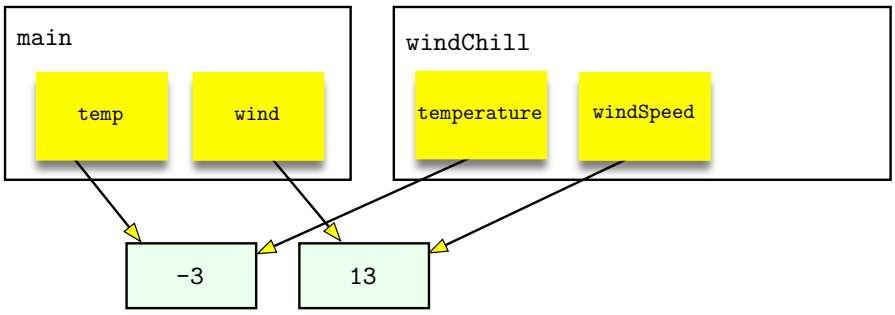
```
def windChill(temperature, windSpeed):
    """ (docstring omitted) """

    print('Local namespace at the start of windChill is\n\t', locals())
    chill = 13.12 + 0.6215 * temperature \
                  + (0.3965 * temperature - 11.37) * windSpeed ** 0.16
    temperature = round(chill)
    print('Local namespace at the end of windChill is\n\t', locals())
    return temperature

def main():
    temp = -3
    wind = 13
    print('Local namespace in main before windChill is\n\t', locals())
    chilly = windChill(temp, wind)
    print('Local namespace in main after windChill is\n\t', locals())
    print('The wind chill is ' + str(chilly) + ' degrees Celsius.')

main()
```

Figure 2.10 The complete wind chill program, with calls to the `locals` function.

```
Local namespace in main before windChill is
        {'temp': -3, 'wind': 13}
The wind chill is -8 degrees Celsius.
```

This is showing us that, at that point in the program, the local namespace in the `main` function consists of two names: `temp`, which is assigned the value `-3`, and `wind`, which is assigned the value `13`, just as we visualized above. The curly braces (`{ }`) around the namespace representation indicate that the namespace is a *dictionary*, another abstract data type in Python. We will explore dictionaries in more detail in Chapter 7.

Returning to the program, when `windChill` is subsequently called from `main`, it is implicitly assigning `temperature = temp` and `windSpeed = wind`, so the picture changes to this:



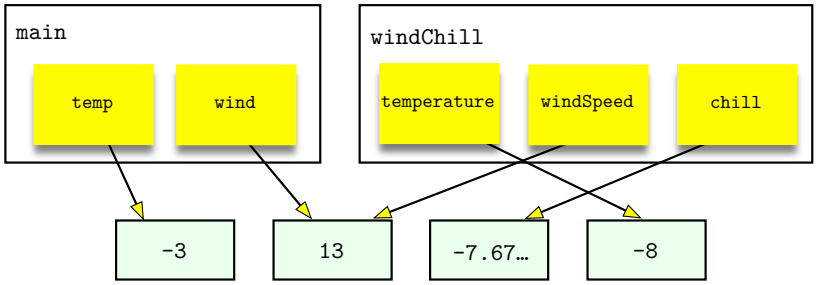
To see all of the namespace changes in the program, insert three more calls to the

`locals` function, as shown in Figure 2.10. Now when we run the program, we see (line numbers added):

```
1 Local namespace in main before windChill is
          {'temp': -3, 'wind': 13}
3 Local namespace at the start of windChill is
          {'temperature': -3, 'windSpeed': 13}
5 Local namespace at the end of windChill is
          {'temperature': -8, 'windSpeed': 13, 'chill': -7.676796032159553}
7 Local namespace in main after windChill is
          {'temp': -3, 'wind': 13, 'chilly': -8}
9 The wind chill is -8 degrees Celsius.
```

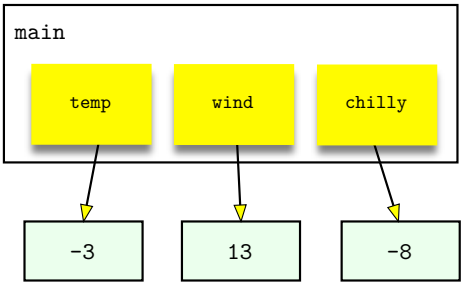
Line 3 above, which corresponds to the preceding "sticky note" illustration, shows us that, at the beginning of the `windChill` function (line 4 in Figure 2.10), the only visible names are `temperature` and `windSpeed`, which have been assigned the values of `temp` and `wind`, respectively. Notice, however, that `temp` and `wind` do not exist inside `windChill`, and there is no direct connection between `temp` and `temperature`, or between `wind` and `windSpeed`; rather they are only indirectly connected through the values to which they are both assigned.

Lines 5–6 in the `windChill` function insert the new name `chill` into the local namespace, assign it the result of the wind chill computation, and reassign the local parameter `temperature` to the rounded wind chill value:



This corresponds to line 5 of the output above.

After the `windChill` function returns −8, the namespace of `windChill`, and all of the local names in that namespace, cease to exist, leaving `temp` and `wind` untouched in the `main` namespace. However, as shown below and in line 7 of the output above, a new name, `chilly`, is created in the `main` namespace and assigned the return value of the `windChill` function:



When the `main` function ends, its local namespace also disappears.

The global namespace

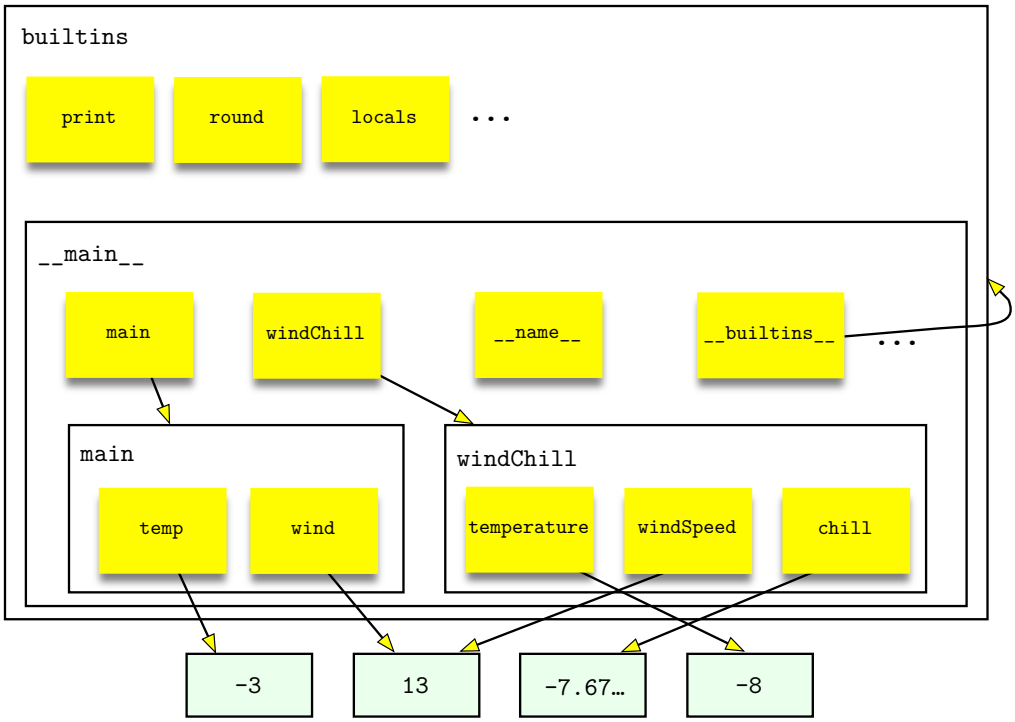
The namespace in which global variable names reside is called the ***global namespace***. We can view the contents of the global namespace by calling the `globals` function. For example, add the following call to `globals` to the end of `main` in our program:

```
print('The global namespace is\n\t', globals())
```

The result will be something like the following (some names are not shown):

```
The global namespace is
        {'__name__': '__main__', '__doc__': None, ...,
         '__builtins__': <module 'builtins' (built-in)>, ...,
         'windChill': <function windChill at 0x10dde8b80>,
         'main': <function main at 0x10dde8c10>}
```

Notice that the only global names that we created are the names of our two functions, `windChill` and `main`. We can think of each of these names as referring to the functions' respective namespaces, as illustrated below (references for some names are omitted):

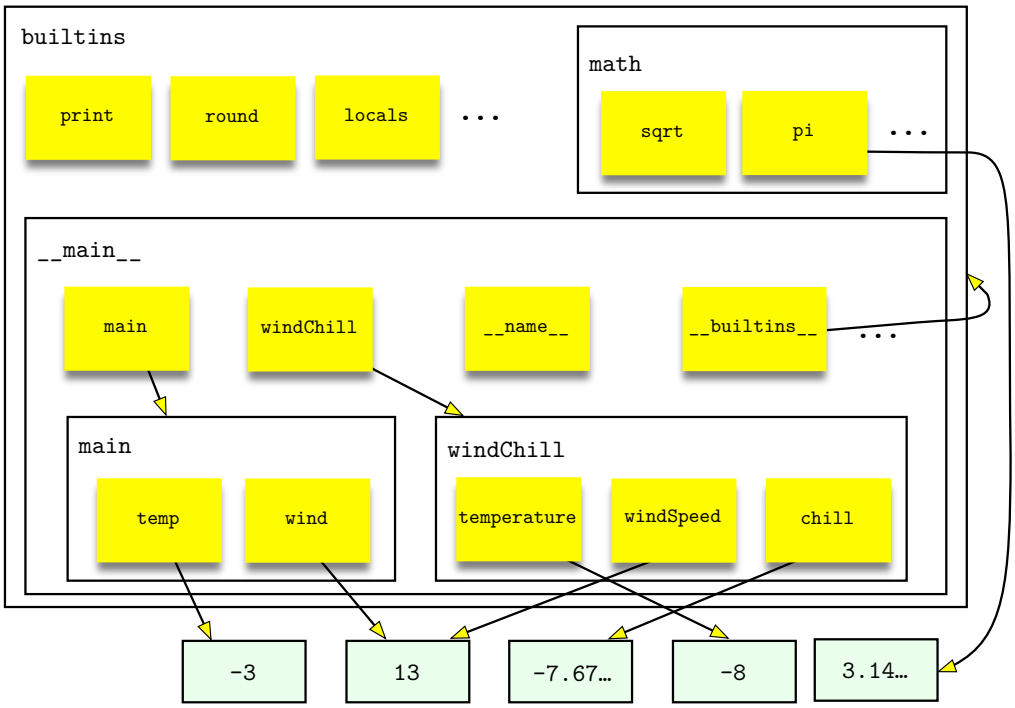


The other names defined in the global namespace are standard names defined in every Python program. The name `__name__` refers to the name of the current module, which, in this case, is `'__main__'` (not to be confused with the `main` function); `__name__` always refers to `'__main__'` when the program is executed directly by the Python interpreter (vs. being imported from another program). The name

`__builtins__` refers to an implicitly imported module that contains all of Python's built-in functions.

As the illustration suggests, we can think of these namespaces as being nested inside each other because names that are not located in a local namespace are sought in enclosing namespaces. For example, when we are in the `main` function and call the function `print`, the Python interpreter first looks in the local namespace for this function name. Not finding it there, it looks in the next outermost namespace, `__main__`. Again, not finding it there, it looks in the `builtins` namespace.

Each module that we import also defines its own namespace. For example, when we import the `math` module with `import math`, a new namespace is created within `builtins`, at the same nesting level as the `__main__` namespace, as illustrated below.



When we preface each of the function names in the `math` module with `math` (e.g., `math.sqrt(7)`), we are telling the Python interpreter to look in the `math` namespace for the function.

Maintaining a mental model like this should help you manage the names that you use in your programs, especially as they become longer.

Exercises

2.6.1* When the `windChill` function in Figure 2.10 is called from `main`, the value of the argument `temp` is assigned to the parameter named `temperature`. Then, in the function, `temperature` is assigned a new value. Does this affect the value of `temp`? Use the pictures in this section to explain your answer.

2.6.2* Exercise 2.5.6 asked you to write a `distance` function to find the distance between two points. Here is that function in a simple but complete program.

```
1  import math
2
3  def distance(x1, y1, x2, y2):
4      dist = math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)
5      return dist
6
7  def main():
8      theDistance = distance(3, 7.25, 9.5, 1)
9      print(theDistance)
10
11 main()
```

(a) Show how to use the `locals` function to print all of the local variable names in the `distance` function just before the function returns. What does the namespace look like?

(b) Show how to use the `globals` function to print the global namespace at the end of the `main` function. Which of the names from the program are in the global namespace?

(c) Insert a statement in the `main` function between lines 8 and 9 to print the local variable `dist`. What happens and why?

2.6.3. Look back at the program in Figure 2.8 on page 73.

(a) In what namespace is the variable `george`. Why?

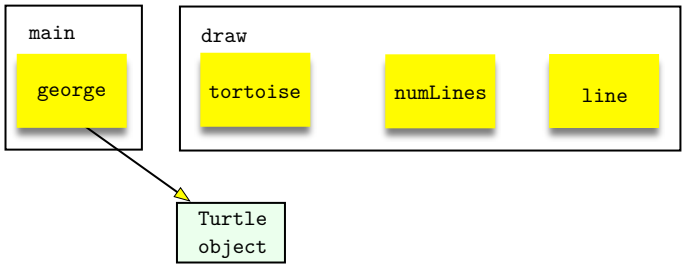
(b) In the `bloom`, `stem`, and `flower` functions, we used a turtle parameter named `tortoise` instead of `george`. Would the program still work if we replaced every instance of `tortoise` with `george`? Explain your answer.

(c) If you made the changes in part (b), the name `george` would exist in two different namespaces while each of the three functions was executing. Explain why. While the `bloom` function is executing, which `george` is being used?

2.6.4. Insert a call to the `locals` function inside the `for` loop in this program. What values is the variable `line` assigned in the loop?

```
import turtle

def draw(tortoise, numLines):
    for line in range(numLines):
        tortoise.up()
        tortoise.goto(line * 10, 0)
        tortoise.down()
        tortoise.goto((numLines - line + 1) * 10, 200)

def main():
    george = turtle.Turtle()
    draw(george, 12)

main()
```

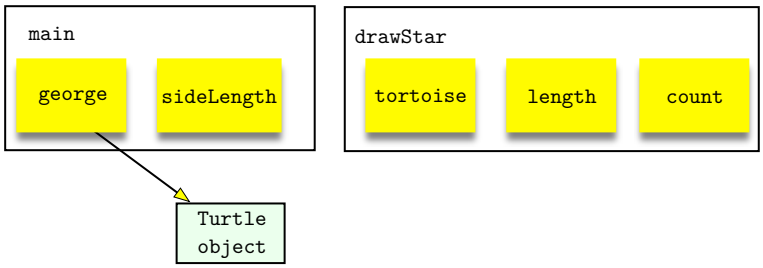
2.6.5* Sketch a picture like that on page 100 depicting the namespaces in the program in the previous exercise just before returning from the `draw` function. Here is a picture to get you started:



2.6.6. Consider the following program:

```
import turtle

def drawStar(tortoise, length):
    for count in range(5):
        tortoise.forward(length)
        tortoise.left(144)

def main():
    george = turtle.Turtle()
    sideLength = 200
    drawStar(george, sideLength)

main()
```

Sketch a picture like that on page 100 depicting the namespaces in this program just before returning from the `drawStar` function. Here is a picture to get you started:



2.6.7. In economics, a demand function gives the price a consumer is willing to pay for an item, given that a particular quantity of that item is available. For example, suppose that in a coffee bean market the demand function is given by

$$D(Q) = 45 - \frac{2.3Q}{1000},$$

where Q is the quantity of available coffee, measured in kilograms, and the returned price is for 1 kg. So, for example, if there are 5000 kg of coffee beans available, the price will be $45 - (2.3)(5000)/1000 = 33.50$ dollars for 1 kg. The following program computes this value.

```
def demand(quantity):
    quantity = quantity / 1000
    return 45 - 2.3 * quantity

def main():
    coffee = 5000
    price = demand(coffee)
    print(price)

main()
```

Sketch a picture like that on page 100 depicting the namespaces in this program just before returning from the `demand` function and also just before returning from the `main` function.

2.6.8. In the program from the previous exercise, change `return 45 - 2.3 * quantity` to `print(45 - 2.3 * quantity)`. How does this change your pictures?

2.6.9. Here is a simple program with the `fleshKincaid` function from the previous section.

```
def fleschKincaid(averageWords, averageSyllables):
    """ (docstring omitted) """

    readingLevel = 0.39 * averageWords \
                 + 11.8 * averageSyllables - 15.59
    return readingLevel

def main():
    theReadingLevel = fleschKincaid(4.0, 1.5)
    print('Local namespace:', locals())
    print('The reading level is ' + str(theReadingLevel) + '.')

main()
```

(a) Run this program. What is printed by the highlighted line above? How does `theReadingLevel` get this value?

(b) In the `fleschKincaid` function, replace `return readingLevel` with `print(readingLevel)`, and run the program again. Now what is printed by the highlighted line above? Why?

2.7 SUMMARY AND FURTHER DISCOVERY

In this chapter, we made progress toward writing more sophisticated programs. The key to successfully solving larger problems is to break the problem into smaller, more manageable pieces, and then treat each of these pieces as an abstract "black box" that you can use to solve the larger problem. There are two types of "black boxes," those that represent things (i.e., data, information) and those that represent actions. A "black box" representing a thing is described by an *abstract data type* (ADT), which contains both hidden data and a set of functions that we can call to access or modify that data. In Python, an ADT is implemented with a *class*,

and instances of a class are called *objects*. The class, such as `Turtle`, to which an object belongs specifies what (hidden) data the object has and what *methods* can be called to access or modify that data. Remember that a class is the "blueprint" for a category of objects, but is not actually an object. We "built" new `Turtle` objects by calling a function with the class' name:

```
george = turtle.Turtle()
```

Once the object is created, we can do things with it by calling its methods, like `george.forward(100)`, without worrying about how it actually works.

A "black box" that performs an action is called a *functional abstraction*. We implement functional abstractions in Python with functions. Earlier in the chapter, we designed functions to draw things in turtle graphics, gradually making them more general (and hence more useful) by adding parameters. We also started using `for` loops to create more interesting iterative algorithms. Later in the chapter, we also looked at how we can add return values to functions, and how to properly think about all of the names that we use in our programs. By breaking our programs up into functions, like breaking up a complex organization into divisions, we can more effectively focus on how to solve the problem at hand.

This increasing complexity becomes easier to manage if you follow the guidelines for structuring and documenting your programs that we laid out in Section 2.4.

Notes for further discovery

The chapter's first epigraph is once again from Donald Knuth, specifically his address after receiving the 1974 Turing award [32]. You can read or watch other Turing award lectures at `http://amturing.acm.org`.

The second epigraph is from Ada Lovelace, considered by many to be the first computer programmer. She was born Ada Byron in England in 1815, the daughter of the Romantic poet Lord Byron. (However, she never knew her father because he left England soon after she was born.) In marriage, Ada acquired the title "Countess of Lovelace," and is now commonly known simply as Ada Lovelace. She was educated in mathematics by several prominent tutors and worked with Charles Babbage, the inventor of two of the first computers, the Difference Engine and the Analytical Engine. Although the Analytical Engine was never actually built, Ada wrote a set of "Notes" about its design, including what many consider to be the first computer program. (The quote is from Note A, page 696.) In her "Notes" she also imagined that future computers would be able to perform tasks far more interesting than arithmetic (like make music). Ada Lovelace died in 1852, at the age of 37.

The giant tortoise named Lonesome George was, sadly, the last surviving member of his subspecies, *Chelonoidis nigra abingdonii*. The giant tortoise named Super Diego is a member of a different subspecies, *Chelonoidis nigra hoodensis*.

The commenting style we use in this book is based on Google's official Python style guide at `https://google.github.io/styleguide/pyguide.html`.